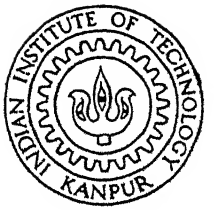
A DEBUGGER FOR THE IITKIX KERNEL

by

ANINDYA CHAKRABORTY



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY, 1990

A DEBUGGER FOR THE IITKIX KERNEL

A Thesis Submitted

in Partial Fulfilment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

by

ANINDYA CHAKRABORTY

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY, 1990

dedicated to
my
parents

CERTIFICATE

This is to certify that the thesis work entitled *"A KERNEL DEBUGGER FOR IITKIX"* has been carried out by Anindya Chakraborty under my supervision, and has not been submitted elsewhere for a degree.

January 1990,
Kanpur

(G.Barua)
Assistant Professor
Deptt. of Computer Sc. a
Engineering
I.I.T Kanpur

ACKNOWLEDGEMENT

I express my earnest gratitude to Dr. G. Barua, my guide and mentor, for his constant help, suggestions and encouragement. I would like to thank my friend P.V. Ramesh for providing some facilities which proved to be useful during my work. I also want to express my thankfullness to all CSE lab staff especially Mr. Pandit for their co-operation at each phase of this project. I thank all my CSE and Hall-IV friends who have made my stay in IIT Kanpur enjoyable.

Anindya Chakraborty

(Anindya Chakraborty)

M. Tech Roll No:- 8811103

ABSTRACT

This report describes the design and development of **KDB : A Kernel Level Debugger For IITKIX**. KDB avoids conventional kernel services during its operation by using its own filesystem, avoiding character device i/o and finally by sitting at the same virtual address space as that of the kernel. This attempt to isolate the debugger from the kernel is to provide better debugging and to make it portable. The debugger will also become insensitive to changes in the kernel. The debugger provides all the common high level debugging facilities and also allows access at the assembly and machine levels. Break_pointing and stepping is achieved with the help of exception processing. The executable file of the kernel, compiled with debug option, is utilised to extract all the high level debugging information. The private filesystem is accessed for providing source line displaying facility.

CONTENTS

CHAPTER – 1

INTRODUCTION

1.1: IITKIX:A UNIX-SYSTEM III COMPATIBLE O/S KERNEL

Few years back a project was taken up by IIT KANPUR Computer science department for developing a UNIX like operating system kernel. The purpose basically was to give the students a feel of how an operating system works and to make them familiar with the art of kernel code generation. The project saw an humble start when an available source code of *UNIX* version 6 for the PDP 11/46 was typed in by a class of 40 students. At that point of time the principal aim was to replace the kernel of a system III UNIX running on a *MC68000* based UPTRON S-32 machine and to test the new kernel with the existing utility programs.

Then onwards the IITKIX has undergone many changes and enhancements through a number of M.Tech projects and short term projects. Now it is fully compatible with UNIX system III and successfully ported on a HCL HORIZON III m/c based on *MC68020* CPU. With conventional UNIX features, the concept of threads which is particularly useful in a multiprocessing environment has also been added. Within a few months a few more enhancements like networking facility and a 4.2BSD compatible filesystem along with a demand paged virtual memory will be coming up.

1.2: AIM OF THIS PROJECT AND MOTIVATION BEHIND IT

A sense of lacking prevailed at each phase of this development project amongst the developers. What they found missing was a proper debugging facility during the phase of actual testing. There was no documentation and source code on the ROM based monitor, used in booting the system and providing a m/c level debugging facility and further, this facility was also found to be very poor. The available source code of kernel was not also error-free.

Inspite of all these difficulties the project has made progress but

the nonavailability of a proper debugging facility has considerably slowed down the pace of developments.

The current project is to bridge the difficulty of actually devloping a software and to get it working. Our aim in this project was to built a kernel level debugger for IITKIX with full fledged symbolic debugging facility along with lower level debugging features.

1.3: KERNEL

A Unix like operating system consists of three layers -

1. The innermost and most privileged layer, kernel .
2. The middle layer, called the shell, is the layer that the user interfaces with. This layer contains a command interpreter which decodes and carries out user commands.
3. The outermost layer contains programming tools like editors, assemblers, compilers, user level debuggers and application programs like accounting packages.

The Kernel conatins a Process scheduler, a Hierarchial file system and mechanisms for processes to Communicate with each other.

The job of the kernel includes process management, memory management, file management and providing communicative and synchronisative primitives. The kernel code can be divided into three parts -

1. The portion written in assembly language. This part deals with the interrupt servicing, interfacing with user programs and the manipulation of runtime environment during context switching. This part traditionally is about 10% of the whole kernel code.
2. The second part implements the file system. It includes device specific driver routines but major portion of it is hardware independent.
3. The last part does the process and memory management and implements user system calls. Some portions of this part is hardware dependent.

The second and third part is mostly written in C language and constitutes a

major part of the kernel code.

The Kernel maintains a significant set of internal data structures mostly in form of tables which are treated as global throughout the kernel code. We mention some of their names (from IITKIX) -

proc table, inode table, file table, mount table, per process open file table, thread table and finally the most significant U-Structure containing valuable informations about the currently running process. It keeps information like saved central processor registers, open file descriptors, accounting information, a pointer to in-core process table entry corresponding to this segment etc.

The U-area lies in the kernel address space and the switching from one process to another is achieved simply by changing the page table entry corresponding to this page.

1.4: KERNEL DEBUGGER

It is essentially a facility through which an operating system designer and implementor can verify the way the kernel is behaving on a particular point of time, for example in case of a system call from a user level process.

The basic purpose of a kernel level debugger is to examine or set a particular internal data structure of the kernel on the occurance of an event. So point of interest of such a debugger is no longer the user process that wants a service the kernel but the kernel servicing it.

1.5: ORIENTATION OF THIS THESIS

In this thesis, Chapter 2 describes various debugging principles, illustrates the concept of debugging levels and also introduces some of the commonly used user level debuggers along with some tools used in kernel debugging in various operating systems.

Chapter 3 will be concerned with the design considerations behind this work. A moderate details of the implementation issues are also presented in that chapter. The portability factors and the development phases are also briefly discussed.

The thesis concludes with a chapter discussing some of the shortcomings of this work and suggesting possible enhancements.

CHAPTER – 2

DEBUGGING PRINCIPLES

2.1: THE TERM DEBUGGING

The most abhored yet almost unavoiable part in the process of software development is the phase of finding and fixing program errors, commonly known as **Bugs**. Program errors can be considered from two perspectives - *Cause* and *Effect*. The goal of **program testing** is to detect errors by discovering their effects, while the goal of **debugging** is to search for the associated cause [MYR78]. Describing in more concrete terms debugging is a two part process; it begins with some indication of the existence of an error (e.g the results of a test run) and it is the activity of -

1. Determining the exact nature and location of the suspected error within the program and
2. Fixing or repairing the error or bug (and hence the term debugging).

In general bugs can be classified into two basic categories of easy bugs and hard bugs. In a typical program most of the bugs found and corrected wihout much pain. But 20% of the bugs becomes very hard to fix as they occur at places where the programmer develops a mind set as how the portion is suppose to work when in fact it works differently.

Debugging as a whole is a hard process as

- It requires repeated switches between intuitive and analytic thought modes and hence creates a psychological pressure.
- Programmers think their programs are behaving in a way which is different from the way it actually behaves.
- Available tools are not adequate in most of the situations.
- Sometimes the program semantics change as the user finds and fixes bugs. This makes it difficult for the user to develop a consistant mode of program behavior [GRA83].

2.2: DEBUGGING METHODOLOGIES

a) Debugging By Brute Force:-

This is the commonest method adopted in program debugging and

is rather inefficient. The popularity of this method can be attributed to the fact that this method is less taxing as it requires little thought. This method can be subcategorised to -

a.1) debugging with a storage dump

a.2) debugging with scattered print statements placed inside the program to determine the program dynamics.

a.3) debugging with automated debugging tools

The first subcategory has the drawbacks like :

1. The difficulty in establishing the correspondance between storage locations and variables in one's source program.
2. Of the massive data available, most is irrevelent.
3. This method gives a static picture of a program but for most errors the program dynamics is very important.
4. Dump is seldom produced at the exact point of error.
5. The lack of describable methodologies for finding the cause of an error by analysing the dump.

Subcategory 2 though better than the first, has the following shortcomings :

1. It is largely a hit-or-miss method rather than one based on thinking.
2. Often results in massive data to analyse.
3. It requires a change in program and hence can mask off the error, alter critical timing relationships or may even introduce new errors.
4. Cost of using it for large systems may be immense and often infeasible on certain types of programs.

Subcategory 3 counts for the whole set of debuggers we see around us. With this one can analyse the dynamics of the program without inserting changes in the program. A common feature of such debugging tools is the ability to set **Breakpoints** causing the program to be suspended on a particular event. This also is a hit-or-miss method and often results in excessive irrevelent data.

The other methodologies involves and depends on human thought processes and hence less popular. they are

b) Debugging By Induction

c) Debugging By Deduction

d) Debugging By Backtracking:-

e) Debugging By Testing:-

Details about these techniques can be found in [MYR78].

2.3: DEBUGGING LEVELS

Generally debugging levels are vaguely categorised as Low Level and High Level debugging with no well defined boundary between them [HAM83]. The level mostly is determined by the user interface to the debugger. Broadly speaking we classify debugging levels as

a) ***Hardware Level Debugging***:- This form of debugging is mostly concerned with signal levels and waveforms. Logic analyser, Digital oscilloscope are the debugging tools at these levels.

b) ***Low Level Software Debugging***:- The techniques involved are memory dumps and interactive examination of memory. Both requires break_pointing. Almost every Unix like operating system supports a snapshot dump facility on address access violations. Today many operating systems also have some variant of DEC Debugging Tape Tools (DDT tools).[HAM83]

c) ***High Level Software Debugging***:- The technique (b) is handled in a formatted manner by the debugging tools at this level. Also the execution history can be shown at source code level. Interactive debuggers can employ type information to display the variables of a program. These can set Break_points based on complex condition parhaps using expressions written in the source language itself.

d) ***Advanced Level Debugging***:- The technique employes advanced tools like a software simulator. It is a computer program that goes through the operating cycle of a computer, keeping track of the contents of registers, flags and memory locations. A typical simulator supports facilities like

a. break_points; b. register & memory dumps;

c. trace; d. setting facility for registers and memory locations.

etc. In real time environment this entire software base of a simulator is often a bottleneck [LEV87].

2.4: ESSENTIAL FEATURES OF A DEBUGGER

a) ***Running*** : Cause the program to start execution under the control of the debugger.

b) ***Single stepping*** : Execute one line of the program and await more commands.

c) ***Position Breakpointing*** : Stop execution of a program at a given point in the program.

d) ***Event Breakpointing*** : Cause the execution of a program to stop when a specified variable takes on a specific value. It is usual to be able to specify that event has to happen in a specified function or procedure to be of relevence.

e) ***Continuing*** : Having stopped at a break_point the user can restart the program where it left off.

f) ***Code Listing*** : Programmers do not always have up-to-date listings, so allowing persual of the code without having to leave the debugger is an useful feature, particularly for placing the positional break_points.

g) ***Stack Trace*** : Displays the functions currently active (the activation tree) and the values of the parameters paassed to each function.

h) ***Printing the Value of a Variable*** : Prints the value of a variable given the symbolic name.

i) ***Changing Value of a Variable*** : The usual method of working involves correcting a bug by editing the source code, recompiling and retesting. It is often useful, having found one bug, to correct the value of the variable and to continue execution.

Most debuggers have the above features and it is these features that are presented to the user in the interface. However, these functions, by themselves, are only mechanisms for manipulating s/w; they do not support the user by stating what is wrong with the program. They are tools that someone can use to probe with. Essentially debuggers assume the user is a person of the analyst/designer/programmer type, being in full managerial control of a relatively small piece of software [WIN88].

2.5 HOW USER LEVEL DEBUGGERS WORK

The user level debuggers available with almost all the current systems among which majority are high level debuggers allowing an user interface where expressions can have a syntax same as that of source languages.

The debuggers can directly extract from the executable file, all the required symbolic information about the source program variables as well as the source code lines with their addresses. As the debugger is normally invoked in the same directory where the source files along with the executable files reside, so displaying of source lines is done through simple file i/o at runtime.

The implementation of break_points and stepping etc. are done through a special system call *'ptrace'* or process tracing. In most cases the debugger spawns a child process and overlays the program to be tested on the child. The *ptrace* system call has a syntax

ptrace(command, pid, address, data)

where the *command* specifies whether to read/write data, resume execution and so on. *pid* gives the process id of the traced process, *address* being the virtual address of interest to the debugger (mostly obtained by a translation of a user typed symbol through symbol table entries) and *data* often provides the numerical value for setting variables in the child's address space.

When executing a *'ptrace'* system call, the kernel verifies that the debugger has a child having the process id *pid* and that the child has declared itself in traced state (done before overlaying the test program). It then uses a global *trace* data structure to transfer data between the two processes. It locks the *trace* data struture to prevent other tracing processes from overwritting it, copies *command*,

address and *data* into the structure, wakes up the sleeping child process and puts it in *ready to run* state and then debugger process sleeps till the child responds. When the child resumes execution (in *kernel mode*), it executes the right trace command, writes its reply in the *trace* structure and awakens the parent (debugger). Depending upon the *command* type the child may re-enter the trace state and wait for a new command or return from handling signals and resumes execution. When the debugger resumes execution the kernel saves the return value supplied by the traced process, unlocks the *trace* structure and returns to the user mode.

This method of implementing a debugger suffers several drawbacks-

1. expensive as it requires four context switches to get information from child.
2. as communicative global data structure (trace) is locked across context switches, multiprocess debugging and more than one simaltaneous debugging sessions on same m/c can be very costly in terms of time.
3. a debugger process has no control over any grandchild in case the child *forks*.
4. if the traced child overlays on itself a different program there is no way the debugger can tally the symbolic information set with the new process as it is read only from the executable file that the debugger has *exec*-ed over its child and not that the child has *exec*-ed over itself.
5. an already running process cannot be traced in *ptrace* of latest available Unix releases.
6. programs doing *setuid* etc. cannot be traced as Unix cannot afford to leave security breeches that can possibly arise in such cases.

Different methods are also suggested, but not yet adopted widely. One method suggested by **Killian** [BAC86] is to access a process as a file though a mounted file system (***'/proc'*** say) and thereby examine or set the processes variables.

2.6 SOME COMMONLY USED USER LEVEL DEBUGGERS :A Brief Survey

a) ***adb*** :- This is an assembly languange debugger first introduced in Unix

Version 7.0. It is one of the first programs adopted when Unix is ported to a new m/c. The process of porting Unix involves m/c specific bugs for which the debugger is essential.

But as a high level debugger *adb* is useless. Most programmers use high level languages and work better with the source they have originally created rather than trying to act as their own compiler. *Adb* has a very extensive command language and hence is difficult to learn [UMNI86].

b) ***sdb*** :- It is a high level debugger provided with Berklely 4.1BSD and Unix system V. It can be used to debug programs written in C, FORTRAN 77 and PASCAL. The command set, like *adb*, is arcane; but the terseness of the command language can be useful for faster learning. Like *adb* there is no scope of event break_pointing. Amongst few of the good facilities worth mentioning, the array subrange in a part of expression to be evaluated (---*a_arr*[1;2].---) or full range (---*a_arr*[*].---) etc. are quite user friendly features [UMNM88].

c) ***cdb*** :- It is a debugger provided with 4.2BSD for C, FORTRAN and assembly programs. Depending on the options installed, *cdb* may be used to debug single or multiple processes which are native or remote to the processor that *cdb* is running on. For future it allows the option to debug an already running process by mentioning the *pid*. It also supports options to override the compiled in address of the U-structure in the kernel space or to override offset of ***u_ar0*** field in U-area. It analyses and create a *(name of executable file).cdb* file containing the necessary symbolic information for it and thereby avoids the overhead of rebuilding the symbol table every time the same version of the object file is tested [UMNI86].

d) ***dbx*** :- Dbx has decended from *pdx* , a debugger by Mark Linton for the Berkley Pascal Interpreter running under Berkley Unix. It is supplied with BSD4.2 and has gained wide popularity because of its easy-to-remember commands and completeness. Aliasing is supported and outputs are better arranged (for e.g if the use asks to show a value of a *"char **"*, then the user really wants to see the string it is pointing at). *Dbx* takes care of these facts [UMNI86].

All these above mentioned debuggers can also handle a ***core file*** which is a core dump produced automatically by the Unix Systems when an illegal address

reference is seen or is produced by a programmer through explicit *signals* .

Coming to a bit more user specific, but advanced set of debuggers, the following must be named -

e) ***dbxtool*** :- It is a window and mouse based debugger for C, Pascal and Fortran 77 running on SUN workstations. Compared to Unix 4.2BSD *dbx* from which it is derived and extended, it has features like ability to debug already running programs, SUN o/s kernel and multiprocess programs.

This debugger is implemented as two processes - a window based front and a slightly modified version of *dbx* connected via a Unix ***pipe***. Hence user can interact even when the process being debugged is running under it before encountering a break. This is possible as the front end process never sleeps. The added interesting features of *dbx* version associated it , are achieved through modification in the object file format (especially the symbol table part) by rewriting the compiler and by extending the *'ptrace'* service of the kernel. Some of these features are mentioned below -

i) ***Walk through stack*** for a particular variable's activation values - a feature particularly effective for debugging recursive routines.

ii) ***Debugging an already running process*** - The 4.2BSD *dbx* supports debugging of live processes and post mortem debugging only for children. This restriction is removed by a provision *attaching* dbx to the process of testing interest and on completion of debugging session again *detaching* it so that it can run freely.

iii) ***Multiprocess debugging*** - Two techniques are generally used. A single debugger can know about more than one process and provide access to all of them; or there can be an instance of the debugger for each process. Here the latter is chosen.

iv) ***Kernel debugging*** - Will be considered in the next section.

The extensions in *'ptrace'* is achieved by adding a field in the process structure in the kernel which keeps the *pid* of the tracing process (the debugger). So it is no longer required to be debugger's child, for *ptrace* to communicate through the above-mentioned *trace* structure [ADA86].

f) ***Joff*** :- This debugger runs on the Bilt terminal and provides a window based user

interface to some AT&T Unix systems by co-ordinating processes on the main CPU with Built processes that manages windows, the mouse and key boards for them. Good features apart from menu driven display methods are to walk through execution stack and to follow data structures comprising of pointers and records.

g) ***DEBUG*** :- DEBUG is a debugger for Fortran, Pascal and C programs running under AEGIS and DOMAIN/IX o/s on Apollo workstations. One of the striking features of this debugger is its ability to debug **optimised code** with some difficulty. It is very hard to debug an optimised code as the one to one correspondance between the variables declared and locations used, or source lines and segment of code produced due to it no longer exists. Moreover the compiler does not reflect these changes in the symbol table formed. Most compilers even disallows optimisation with debugging option set (like Unix 4.2Bsd *cc compiler* in HORIZON III). But Apollo's *DOMAIN cc compiler* supports different user selectable optimisation levels.

DEBUG works with little inconsistancy for optimisations like *cross jumpimg, dead code elimination, assignment merging, common subexpression elimination etc.* But optimisations affecting allocation of space for a user declared variable (Like putting local variables or global variables in a loop in register) or optimisations that reorders instructions in the code space, renders DEBUG ineffective in displaying values of affected variables and source lines.

DEBUG is a fairly high level debugger with facilities like -

- keeping a hit count and there by skipping a particular break_point a user specified number of times.
- defining debugger variables
- jump to arbitrary levels etc [APMN87].

2.7 HOW KERNEL LEVEL DEBUGGING IS DIFFERENT FROM USER LEVEL DEBUGGING

Though the kernel level debuggers are often implemented as an extension of user level debuggers, this restricts the debugging facility within a narrow range of operation as the debugger itself uses call to kernel like *'ptrace'* , screen display etc.

In principle a kernel debugger should be totally independant of the kernel

and should be self-sufficient in handling the hardware resources required by it. It should not rely on any system specific concepts like process, file etc.; because such dependancy may render the kernel debugger useless during porting phase of the kernel to a different m/c.

We express some of the design aspects and constraints that we felt our kernel debugger should provide -

i) As we are concerned with Unix like kernel, we felt that the only language the debugger should take care of, is C.

ii) A full fledged high level debugger may enforce some rigidity in the debugger display, so a total reflection of assembly as well as byte level view is also important.

iii) Effort should be put to make the debugger independent of the kernel services at least during the debugging session. We imposed no concept of process on the debugger, also disk i/o on kernel's *root* and *mounted* file system is totally avoided during debugging session.

iv) No concept of terminal (tty) is imposed on the debugger to avoid character device i/o completely.

2.8 THE APPROACH FOR KERNEL DEBUGGING ADOPTED BY THE PREVIOUS *IITKIX* DEVELOPERS

As mentioned in Chapter 1, earlier IITKIX developers always found actual testing phase very painstaking. To get rid of the uncertainity and the risk involved in testing a freshly written kernel code on actual m/c, they almost always have gone for a first phase of simulation (mimicing hardware *traps* through Unix software *signals*) and have then taken up the job of actual testing.

In UPTRON m/c debugging was done through a very primitive debugger written as different code segments in the kernel itself. The debugger was at m/c level, and was used to be invoked through motivated statements in the ***icode*** (which is executed by ***'init'*** process i.e process 1). In general the programs for testing the kernel utilities were also written in *icode* only.

In later projects a kernel version of C ***printf*** routine was written and was

used by interspersed 'print' statements in the kernel code itself. Note that there was no way of stopping the continuously arriving informations on the console.

2.9 SOME EXISTING KERNEL DEBUGGING FACILITIES

a) ***adb*** :- Though not yet marketed, the recent version of *adb* supports a kernel debugging option (-k). Unfortunately no details are available in this regard.

b) ***System Debugging Facilities in BSD4.3*** :- The most commonly used facility is the *"crash dump"*, a copy of the memory that is saved on a secondary storage by the kernel in the event of a catastrophic failure. Crash dumps are created by the ***"doadamp( )"*** routines. They occur if a *'reboot'* system call is made with the *RB_DUMP* flag set or if the system encounters an unrecoverable , unexpected error. The *"doadump( )"* routine disables virtual memory translation, raises the processor priority level to its highest value to block out all device interrupts and then invokes the ***"dumpsys( )"*** routine to write the contents of physical memory to secondary storage. It is something like a core dump for a user process where only its virtual memory contents are dumped. The precise location of crush dump is configurable; most systems place the information at the back of *primary swap partition*. The device drivers dump entry point is used to do this operation.

A crash dump is retrieved from its location on disk after the system is rebooted and the file systems have been checked. The ***/etc/savecore*** programs exists solely for this purpose. It creates a file into which the crash dump image is copied. *Savecore* also makes a copy of kernel load image, ***/vmunix*** for use in debugging. Crash dumps can be examined with either of the 4.3Bsd debugging programs *adb* or *dbx*. Both debuggers can also be used to examine and modify a running system through the ***/dev/kmem*** special file which allows access to the memory area containing kernel code and data [LEF88].

c) ***DBXtool for Kernel Debugging*** :- When debugging the kernel the *dbx* part in it uses the page maps within the kernel (or, the core image) to map addresses. The **PROC** command allows the user to specify which process's U-structure is mapped in kernel's U-area. Therefore the user can use the kernel stack trace for any active process. The debugger is made immune to changes in the kernel internal data

structures. Most Unix debuggers need to know the format of the *user* structure to read and write a process's registers. If the offset of the stored register values within the *user* structure (*u_ar0*) changes, these debuggers must be re-compiled. Here **GETREG** and **SETREG** option in extended *'ptrace'* system call brings this independancy.

A set of new symbols are added to free the *dbx* from having to know the kernel's internals. The symbols can tell *dbx*- number of pages in the U-area, the size of a page, the base address of kernel and the size of virtual memory. This allows *dbx* to be used on kernels for several different processors and memory architecture. To find process and U-area structures associated with a process, *dbx* uses definition of these structures contained in the debugger symbol table associated with the kernel's executable file (generally resides at the root '/') [ADA86].

d) ***MACH Kernel Debugger*** :- MACH is a multiprocessor operating system kernel and environment under develoopment at Carnegie Mellon University. The MACH kernel has a built-in kernel debugger based on *adb* (this version currently only works under *Vaxen*). All *adb* commands are implemented including support for break_points, single instruction stepping, stack tracing and symbol table translation.

In order to aid debugging, as well as study the performance of the kernel, the MACH debugger also supports functions not available in *adb*; for e.g -

i) *enhanched stack traces* - stack traces may contain the value of local variables and registers for each stack frame.

ii) *call/return trace support* - single stepping may continue without intervention until the next call or return instruction.

iii) *instruction counting* - the number of instructions executed between regions of code may be counted.

During the implementation of the system these features have proven to be invaluable in both debugging and performance tuning [ACC86].

CHAPTER – 3

DESIGN CONSIDERATIONS AND IMPLEMENTATION DETAILS

3.1 PREAMBLE

Our goal was to build a kernel level debugger which resembles a C source level debugger with all assembly and low level debugging options available at the user interface. But the basic difference with any user level debugger or any previously discussed kernel level debugger (which are merely extensions of the user level debugging concept) lies in the fact that it must not impose any dependency on the operating system operations. Let us now explore what a C source level debugger is supposed to do and what are the possible areas where such a debugger may explicitly need operating system services.

3.2 POSSIBLE KERNEL SERVICES NEEDED BY A DEBUGGER

A C source level debugger supporting a minimal set of debugging facilities must provide a set of commands for achieving tasks discussed in section 2.4.

If one runs the debugger in the user mode and just provides some extensions to it for kernel debugging then for almost all the tasks except the symbol table translation, the debugger needs the help of the o/s kernel which itself is under testing. The help needed may be in the form of -

i) A system call to achieve break/steps.

ii) A system call to map the virtual address of the process under supervision to its own virtual address.

iii) A system call to do file i/o on source files so that the source code lines can be shown during a debugging session for the benefit of the user.

iv) system calls to perform i/o on terminals for the sake of interaction with the user.

All these will actually limit the spectrum of debugging ability of the debugger. Furthermore a user level process by definition has a direct dependency on the o/s (which is under supervision in this case) in terms of memory, process and all other

resource management. So it may lead to a circular problem and the debugger's flawless operation can be questioned at times. Another potential problem in such a technique is the inability of the debugger to get service whenever it desires. Being a user process it has to wait for its turn in terms of allocation of kernel resources needed by it. This may lead to a change in actual orientation of kernel data structures with and without the debugger when the same set of events are taking place. This certainly will mask off or aggravate the effect of bugs in kernel code.

All these lead us to attack this problem in a completely new way. Though the basic debugging technique we adopted for achieving our basic goals were used by the previous generation debuggers quite often, the way these techniques have been incorporated into the kernel is new. We finally came up with ***KDB*** : *the kernel debugger for IITKIX*.

3.3 HOW KDB AVOIDS USE OF KERNEL SERVICES

As it is clear from the previous section we cannot afford to have KDB to run as a user process, it is designed to be a kernel process, in fact a process running in the context of the currently running process. The avoidance of kernel services were ensured by taking the following decisions -

1. So far as interaction with the user is concerned, it should be achieved without any asynchronous i/o. It is achieved by two assembly functions ***'sioputchar'*** and ***'siogetchar'*** which uses the console as a serial port connected via a *RS232* line. So *no tty concept is involved*.

2. In the implementation of break/step, the debugger has the full control over the gamut of kernel code and can explicitly modify the memory content at a desired place using *MMU* hardware only and without any kernel intervention.

3. For displaying source lines of the C code, KDB does not use the kernel file system services. It implements its own file system.

4. In block device i/o-s a tradeoff between speed and avoidance of kernel routines are achieved. During initialisation phase of KDB, it uses high level

block device i/o routines of kernels for faster initialisation. As this phase takes place before any real operating system task begins (see section 3.4.2), the tradeoff is justified.

However during interactive phase, the lowest level block device i/o and device driver routines are relied upon by the debugger and thus avoids any modification of internal data structures of the kernel during this phase. At this phase only i/o from KDB's private file system is allowed for debugging purpose.

5. To ensure a complete insensitivity to kernel data structures, the debugger does not use any kernel globals and structures except a set of *buffer* structures.

3.4 THE MAJOR PARTS OF KDB AND SOME IMPORTANT IMPLEMENTATION ISSUES

3.4.1 The Environment

KDB for its successful initialisation needs the compiled code of IITKIX kernel to be resident at the *root* of the IITKIX file system. The code should be compiled with a debug (*-g*) option to extract the full debugging capability of the debugger.

KDB further needs a private file system (format explained later) containing the C source files used for developing the kernel. This filesystem should be resident on a disk partition defined in the *'/dev '* directory of IITKIX as a block device special file.

It is worth mentioning that there is a sharp difference in the woking principle of the two phases - *initialisation* and *interactive* phase. In the initialisation phase the stress is on the speed of the initialisation with a slight tradeoff with space saving whereas the second phase stresses on the isolation of kernel operation with that of KDB.

3.4.2 How and Where KDB gets control for initialising its Private Data Structures

IITKIX like conventional Unix systems boots up through the bootstrap loader loading it and leaving control to the kernel code (loaded by it) at the entry point.

The start routine initialises the virtual memory tables for both itself and for the rest of context slots which will later be occupied by user level processes. Then the control passes to the *'main'* routine which initialises the interrupt service tables and various other internal data structures. It also defines itself as *'process 0'* and then spawns a child process commonly known as *'init'*. Process 0 then loads the *'icode'* stored in static data area of the kernel to the code area of *'init'* and goes into an infinite scheduling loop (*'sched '*) to become the *swapper* process. The *init* ,generally, executes the *'/etc/init '* executable file which spawns child to execute *'gtty '* and finally *'login'* shells on different terminals.

KDB starts initialising itself when process 0 is defined and the kernel is going to spawn the *'init'* process. It must be made very clear thet the debugger is not compiled and kept as a separate utility. On the otherhand it is a set of files which is to be complied with other kernel source/object files. So KDB sits at the same virtual addess space as the kernel and the initialisation starts only when an explicit function call is made from the kernel *'main'*.

Though a subset of debugging commands is made available to the user when KDB stops to interact immediatly after its initialisation, all a user is expected to do at that phase is to set a group of break_points and exit the initialisation phase to continue with the booting. The actual interactive phase of kdb starts after a valid break.

3.4.3 The Modules of KDB

KDB can be viewed as four major parts integrated together. They are

i) the break / step manager

ii) the symbol table information extractor

iii) the disassembler

iv) the user interface

A description of the design issues and implementation steps for each of the above-stated modules is provided below.

3.4.3.1 The Break/Step Manager

KDB works on the principle of *self-code modification*, that is, it modifies the

code at the location where a break_point is to be inserted, which sits in the same virtual address space as the KDB's code.

A break is implemented by replacing the code residing at a specified addess by a *trap instruction* and stepping through the code is achieved by *setting the trace bit* of the processor's status register. Before explaining the manager any further we must briefly introduce the way *MC68020* (the processor on which IITKIX currently runs) behaves in these cases of instruction trap and set trace bit in status register.

The MC68K family of microprocessor has two modes of operations, namely *user* and *supervisor* mode. Its assembly instruction set contains a set of privileged instructions usable only in the latter mode and allows a sofware access to data and registers that are associated with task scheduling and interrupt handling. When the MC68K is reset using *RESET*, it starts operating in the supervisory mode. The supervisor can change the CPU mode to user mode but the opposite naturally is not possible. The only way for the processor to change from user mode to supervisory mode is through an **exception**. The exceptions fall in two broad categories -

1. The exceptions that *originate from within the CPU* and
2. The exceptions that *come from outside the CPU*. This category includes -
   - 2.a) interrupts (mostly from system clock and peripherals).
   - 2.b) bus errors (from memory management unit or MMU).
   - 2.c) system reset

Our interest lies in the first category of the exceptions, the range of which includes -

- odd addressing errors
- illegal and trap instructions (★)
- illegal operations
- invalid use of coprocessor
- privilege violation
- exception tracing(★)

The exception vector table occupies 512 16bit words of memory and is set up by the *'main'* of the kernel. It provides the address of the exception handler to jump to,

when the CPU faces an exception. When one exception occurs while another exception is still pending, the CPU arbitrates according to the priorities assigned to various exception groups.

The MC68K processor can be made to operate at any one of the *eight* levels of priorities by varying the 3 *interrupt mask bits* in status register. If the CPU priority is equal to or greater than the interrupt level, the CPU ignores the interrupt and continues execution [LEV87].

We now justify our choice of trap and trace. So far as implementation of stepping through the code is concerned, tracing is almost an obvious choice. In MC68K processor status register if the trace bit is set, the CPU executes an instruction. But before executing the next instruction, it traps through the *trace vector* to an exception handler. When the CPU begins exception handling , it automatically disables the trace bit so that exception handler can run unhindered. Though our current host processor allows more selective forms of tracing, they are not used for portability issues discussed later. In kernel code varying of a processor's priority (*'spl'* routines) is a very frequently encountered operation. We must not restrict the user from stepping through these regions of the code. A fairly high priority of trace trap also ensures this fact. A **format $2** stack frame is produced on trace trap (Fig 3.1)

So far as the implementation of break_pointing facility is concerned choice of *instruction trap* for portability issues are discussed later. In the MC68K, 15 unconditional trap types are allowed. In IITKIX *trap #0* is used as a *syscall* interface. We used *trap #2* (will be addressed as ***DEBUGtrap*** in this thesis) for breakpointing. When a user declares a break_point by mentioning some information that corresponds to a valid code addess, the part of the instruction beginning at that address is replaced by the trap instruction. As in the MC68K all instructions are *word aligned* this replacement can never affect more than one instruction. This is because of the fact that *"trap #n"* instructions occupies only one word in memory.

When in the course of instruction execution the program counter value becomes equal to that address where a break is set, the CPU after fetching, decoding and executing the instruction finds it to be a trap and begins exception processing. After the steps mentioned earlier in this subsection, KDB gets control

as an exception handler and starts interacting with the user. In MC68020 CPU a **format $0** stack frame is produced in such traps (Fig 3.2).

In KDB a break_point is described in a structure having the following fields :-

a) An *Address* field - Keeps track of the actual location of the break.

b) A *Code* field - Contains the code that is supposed to be resident at the break_point location if their was no KDB intervention.

c) A *Dirty* bit - Representing wheather the code at the break_point location is modified by KDB or not.

d) Some bits representing the *Status* of the break_point - Wheather the break_point is freed , deleted , entered or a stepping is being initiated from the break_point etc.

e) Two fields providing *Line number* and *File identity* of the location where the break_point is placed.

f) A *pointer to a member of the address hash table* - So that the nlist pointed by that member can uniquly represent the routine in which the break is placed irrespective of the high level language considerations.

Clearly break_point in the part of the kernel code used by KDB and in the KDB routine itself may cause an infinite loop of traps and corresponding exception processing without any return. As soon as KDB gets control during a trap or trace exception processing the processor priority is raised to highest level to block out all interrupts except the non-maskable interrupt. During KDB's service it only lowers the processor priority when disk i/o is needed to be initiated for displaying the source lines. The command set hence provides an option to disable the automatic line displaying facility on each halt. Thus a user can also inspect the critical region of kernel's code without any hindrance. The consistancy in the critical region is guaranteed as unless disk i/o is done during KDB's scope of operation no interrupt can disrupt the processing. Note that for disassembly KDB needs to consult the memory only. After KDB reliquishes control the processor priority at the point of DEBUGtrap or trace trap will be restored.

KDB needs to remember some private variables' value until the immediatly next trace exception after finishing a valid break processing (DEBUGtrap exception) appears. Hence a mutual exclusion flag is essential. This is set whenever KDB gets control after a step and reset when KDB relinquishes control back to the normal kernel activity. In case of a break, however, the mutual exclusion flag is only reset

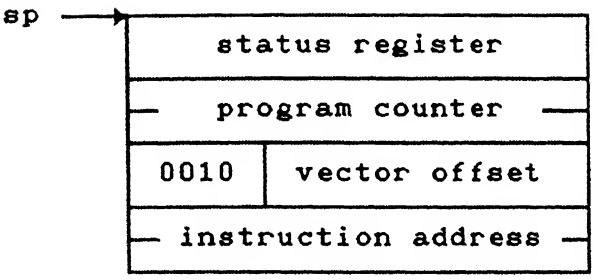


FIG 3.1
Format $2
stack frame

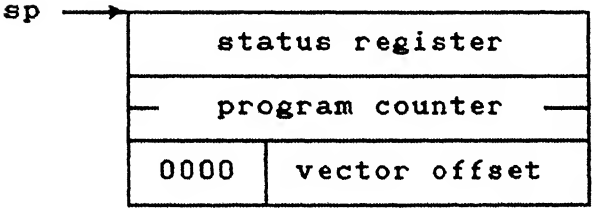


FIG 3.2
Format $0
stack frame

during the immediately next trace exception. This trace trap occurs even when the user decides to continue after a valid break. This is due to the fact that, it is during this period only, when the KDB's break/step manager writes back the DEBUGtrap instruction which was earlier replaced by the original code during DEBUGtrap exception service, at the break_point address. This ensures that if any interrupt occuring in between the two said events causes another break during its service, KDB's variables remain consistent.

Another significant point to be made is that the clock service is aborted whenever the mutual exclusion flag is found ON. This should be done as process switching should not take place when the debugger has control.

KDB's break/step manager utilises the DEBUGtrap and trace exception processing which needs a supervisory stack for the hardware to dump information at the begining of exception processing. So while linestepping a function eventually calls for a context switch (for typical reasons in multiprogrammed environments), the kernel page frame corresponding to the U-area page of the current process is mapped to a page of another processes U-area. Thus the new process gets control from some other region in kernel's code, depending on the pushed value of the program counter when the CPU burst was snatched away from the context of this process. As the supervisory stack is placed in the same page as the U-area page, a trace exception when the U-area page is being redefined, will cause abnormal results as the corresponding MMU and TLB entries are not properly defined at that phase. We avioded this anomaly by keeping track of the *next executable line number* whenever an user is performing linewise stepping. If any function call within the kernel code eventually leads to a context switch, a break_point is placed automatically (if already not present) at the next linenumber and the kernel is allowed to continue with its normal execution. If one explicitly wants to check the state of the kernel while it is inside context switch routine, one can do it by putting explicit break_point at the relevent code or by entering the routine by assembly wise stepping.

3.4.3.2 SYMBOL TABLE INFORMATION EXTRACTOR

The aim of this unit is to built a database comprising of all the symbolic information available in the symbol table part of an executable file and to organise

them in such a way so that the search overhead and the number of places to look into for uniquely identifying an identifier (especially variable) is low.

3.4.3.2.1 **DESIGN ISSUES**

a) FROM THE VIEWPOINT OF C LANGUAGE

C bases the interpretation of an identifier upon two attributes of the identifier, its ***storage class*** and its ***type***. The storage class determines the location and the lifetime of the storage associated with an identifier. The type determines the meaning of the values found in the identifier's storage.

C allows five declarable class specifiers; namely -

1. auto *2. static* *3. extern*

4. register *5. typedef*

Storages of auto (automatic) types are local to each invocation of a block and discarded upon exit from the block. Static variables on the other hand are local to a block but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values through-out the execution of the entire program. Register variables are (if possible) stored in fast registers of the m/c [KEN88].

So far as the type of a storage is concerned, C supports several basic types. Apart from these fundamental types, there is a conceptually infinite class of **derived types** constructed from the fundamental types in the following way.

- arrays of objects of most types;
- functions which return objects of a given type;
- pointers to objects of a given type;
- structures containing a sequence of objects of various types;
- unions capable of containing any one of several objects of variable types.

C language allows two kinds of type specifiers namely :-

i) BASIC TYPES :- 1. *integer*, 2. *char*, 3. *long*, 4.*short*, 5. *unsigned char*, 6. *unsigned short*, 7.*unsigned long*, 8.*unsigned int*, 9. *float*, 10. *double*, 11. *void*.

ii) USER SPECIFIED TYPES :-

1. *struct-or-union specifier.*
2. *typedef name.*

As we are mostly concerned with choosing a proper way of representing every declared C object's storage, the concept of storage class and types must be implicitly or explicitly retained in our design.

b) FROM C COMPILER'S POINT OF VIEW

This part narrates the basic technique adopted by the C compilers to allot and describe an identifier's storage in the symbolic information produced.
All the auto variables are allocated on the stack. The passing of parameters is the responsibility of the caller of a function, whereas in case of locals declared in the function body, the called function defines storage for them after being activated. The frame pointer marks the boundary between the region of caller's responsibility and the region tackled by the called function. The register variables if possible are allotted space in the registers otherwise in the stack. If the register variable deals with the lvalue (i.e pointer or pointer to pointer etc.) of an identifier, the address registers are allotted for them whereas the basic type objects are accommodated in the data registers. C compiler in HORIZON III uses address registers a3 onwards and data registers a7 downwards for register variables. The return values if the size matches, is returned through d0 register. On failure space is provided in the static data area for returning values. The static variables like all external globals are allotted storage in static data region but during translation the compiler enforces the scope rules in their use [UNI86].

Regarding the description supplied by the compiler in the symbol and string table area for the declared identifiers, the following points are worth mentioning.

i) As almost all the compilers allows the provision for separately compiling different source files and finally linking the object files produced thereby, the symbolic description part in an executable file also reflects this independence. Almost all the information (like description/type of an C object, its scope etc) except the final address (known after linking) are generated

during parsing time, hence for each source file the symbolic description of all the identifiers in it are unique and valid within the range of that file only. So in the final merged symbol table found in the executable file, these mutually exclusive sets of symbolic entries appear in the order in which the linking of the object files are done. For example if the object file due to the source file A is linked before the object file produced from a source file B, in the final symbol table the set of symbolic entries due to declarations in file A will be encountered before the set of symbolic entries signifying declarations in file B as one travels down the string table.

ii) A set of unique typenumbers are allotted and used for a particular file. The typenumbers are assigned to a particular type specification, which may be either implicit to C compiler or explicitly provided by the user. These type numbers are completely internally gererated by the compiler as it encounters enumerations in a file during parsing. Across the files these typenumbers are no longer unique as the compiler resets the typenumbers allotted, after the translation in each file is over.

iii) The first 12 numbers (1-12) are always assigned to 11 basic types (see subsection 3.4.3.2.1(a)) with the last number assigned to a dummy ("???") type. Apart from the user specified types, typenumbers are also allotted internally to all the derived types found inside or with a declaration (not necessarily of type specifier but also of any other kind like variable declaration). This happens in spite of the fact that these declarations do not have any explicit type specification attached with them. The derived types may occur in a variable declaration or inside a type specification itself. This implicit assignment of typenumbers is done to avoid redefinition of that derived type for each of its future occurence in other declarations of the same file. For example, in case of a user declared type specification like-

```
struct xxx {
        int *x1;
        int *x2;
}
```

a typenumber will be assigned to the structure specifier 'xxx' as expected, but also another typenumber will be assigned to the derived type 'int *' qualifying the member x1 of the type 'xxx' provided a typenumber has not already been

assigned to the derived type 'int *' due to some other declaration appearing somewhere before in the current file. Now x2 can be qualified by internally assigned typenumber of 'int *' with no need of redefining it in the description related with x2. This process resembles human effort of reducing repeaded definition of a much used derived type, by declaring *'typedef'*-s. This also reduces the range of typenumbers used in a particular file.

We would like to repeat that a typenumber signifying a particular type is meaningless beyond the range of symbolic information set produced out of the concerned source file.

iv) In the set of symbolic information due to a source file, the sequence of symbolic descriptions directly follows the syntactic sequence of declarations in the file. An elaborate discussion regarding this can be found in Appendix-A.

c) OUR POINT OF VIEW

We, in our design of the symbol table information extractor are more concerned with the savings of storage space and initialisation time. Our design should be able to represent the entire set of type specifiers , variables and other C objects declared and also should retain all the information about them which the user may want to retrieve. We felt the necessity of a global database for such an representation as unlike the C compiler, we have all the required information for the entire set of C objects of the target program at our disposal. This globalisation is expected to save time as repeated analysis of the same C objects appearing in different file's information set, need not be fully done. This also saves space and helps in overcoming some anomalies arising in conventional debuggers like DBX.

We felt that there is no need of retaining blockwise scope information for variables and functions. C being a language with static scope, the compiler is responsible for enforcing the scope restrictions and the violation of scope need not be detected at run time. So we simplified storage classes into two broad categories, namely -

- **Globals** - the variables that are allocated in the data region of memory.
- **Locals** - the variables which gets a storage space on stack or registers when activated.

So only one boundary is needed to be recognised to differentiate these two storage classes. As an obvious choice that boundary is taken to be the functions. A block, to KDB, means the body of the function the block syntactically belongs to; whereas any declaration outside the boundaries of of several declared function bodies are taken as global and thus kept in the global database. The boudaries of files (as sometimes meaningful to 'static' declarations) are ignored.

Extraction of storage class information became very straight-forward with this simplification. The *typedef* specifier is only a storage class specifier for syntactic convenience and hence not of our concern. In case of *extern* there must be an external definition for the given identifier somewhere outside the file or function in which they are defined. If any violation is found a linkage time error will be indicated. So we have not extracted the information for this type also. So far as the *register* variables are concerned, we have made an assumption that this type is only meaningful as parameters or local automatic variables. With no file boundary restriction static declarations are synonymous to globals with infinite scope.

The type specifiers are arranged so that a particular typenumber can uniquely qualify a type (explicitly declared or implicitly assumed) over the entire database. This led us to introduce the idea of **REPEAT** types which signifies that the entry corresponding to this typenumber is a repetation of an already defined type description stored at some other identifiable place in the database. Apart from this type we introduced -

i. basic, ii. array, iii. function , iv. pointer, v. structure and *vi. union* types.

A valid typenumber can only stand for one of these seven types.

Our design must not restrict itself only to high level language considerations like scope and types . In fact it must be layered properly so that all low level aspects can be accessed by the user if desired. At low level each declaration (arising out of the assembly or C source files) is either a member of the code or a a member of the data regions. The text routines are the constituents of text segments whereas globals and absolutes are members of data segments. Both are well described by the two similar but mutual exclusive sets of *nlist*-s. These

sets of *nlists* are arranged in two disjoint planes (representing text and data regions) and above them the higher level attributes are kept where ever applicable.

3.4.3.2.2 **IMPLEMENTATION ISSUES**

a) LOW LEVEL

At this level, the central implementation criterion is to reduce search overheads which is in general a significant task done most of the time in a debugger. A kernel level debugger must also be well equipped for quick searching in an addess space as big as that of a kernel. A typical UNIX like operating system can have as many as 500 globals and numerous functions. A typical code size of 15000 C lines is expected along with different lower level routines. No doubt the search overheads must be reduced with great care. At once the choice of hash lists becomes very obvious. But we must also consider what can be the key for searching for the symbolic information units consisting of information about each text or data segment symbol.

The debugger may search a location by name as well as by address. To see a value of a variable or to start disassembly from the begining of a particular routine the search of the first kind has to be activated. On the other hand, rather than producing a disassembly of the form

jsr 0x1000B1AC or
movl 0x108001D4 , d0

the following would be desirable.

jsr _ialloc or
movl _u+#1D4, d0

In these cases a search of the second kind is required. KDB keeps text and data region *nlist*-s in two identical yet mutually exclusive set of structures. Each *nlist* information is inserted in this domain incrementally by hashing on its name and address. But the nature of search demands something more than a simple hashed structure. In case of search by address, it may be required to search for an nlist in whose range of which the address belongs to, similar is true for the data region to provide a user friendly disassembly. For example if the code due to a C function '_a' (as it represented in assembly format) occupies an address range starting from α and ending at address β then a search with address γ ($\alpha \leq \gamma \leq \beta$) must find out the *nlist* signifying the routine '_a' so as to be able to represent the address γ

symbolically in the form "_a+(Y−α)". There is no known hashing algorithm which can ensure that any arbitrary address will always hash to map into the bucket where the desired *nlist* lies. · Also we have to built the hash list incrementally with receipt of any *nlist* information which demands an entry in this hash list. Our design and hash functions are deviced such that even if an address hashes to a bucket not containing the desired *nlist*, the overhead of finding the desired entry starting from the bucket is considerably less than a linear search. As the hashing by name guarantees, the availability of a desired *nlist* in the bucket it hashes to, the hashing function can be straight forward. But in case of hashing by address, the function must guarantee considerable insensitivity to the number of lower bits of address which usually varies in the span of a text/data segment symbol. Expectedly a data segment symbol's address range is less than that of a text segment symbol, so one of two different hashing functions are applied depending on whether the address in consideration lies in the text or data region.

The nlists in a particular bucket are kept sorted in terms of address in the address hash list. To aid the search all the members of the address hash list are connected throughout in a doubly linked list so that the *nlists* they point to are sorted in the order of increasing address. They also contains a size field signifying the range of the address space occupied by that symbolic unit. Fig 3.3 gives a pictorial representation of the orientation of *nlists* in a particular region (data/text). We also present the algorithm of searching an nlist information when the key is an arbitrary address (Fig 3.4). The overhead of building such a orientation sorted in terms of address is almost nil as the addresses assigned to data/text segment entries monotonically increases both within and across files. Hence the symbol table extractor also comes across those entries in the same increasing address order as it scans down the string table.

In this algorithm if an address directly corresponds to a symbolic start address the condition of 3rd *if* statement will be statisfied on the 1st invocation of *'search_bucket'*. The two top level while loops takes care of a sparse distribution of the symbolic units in the hash table which may arise due to a small code size resulting in a moderate number of text and data segment symbols (Note that the hash list is staticlly allocated and has no way to guess the size of symbolic information). However in a Kernel code this scenario will never arise due to a huge set of symbols. In general a search will follow the forward or backward

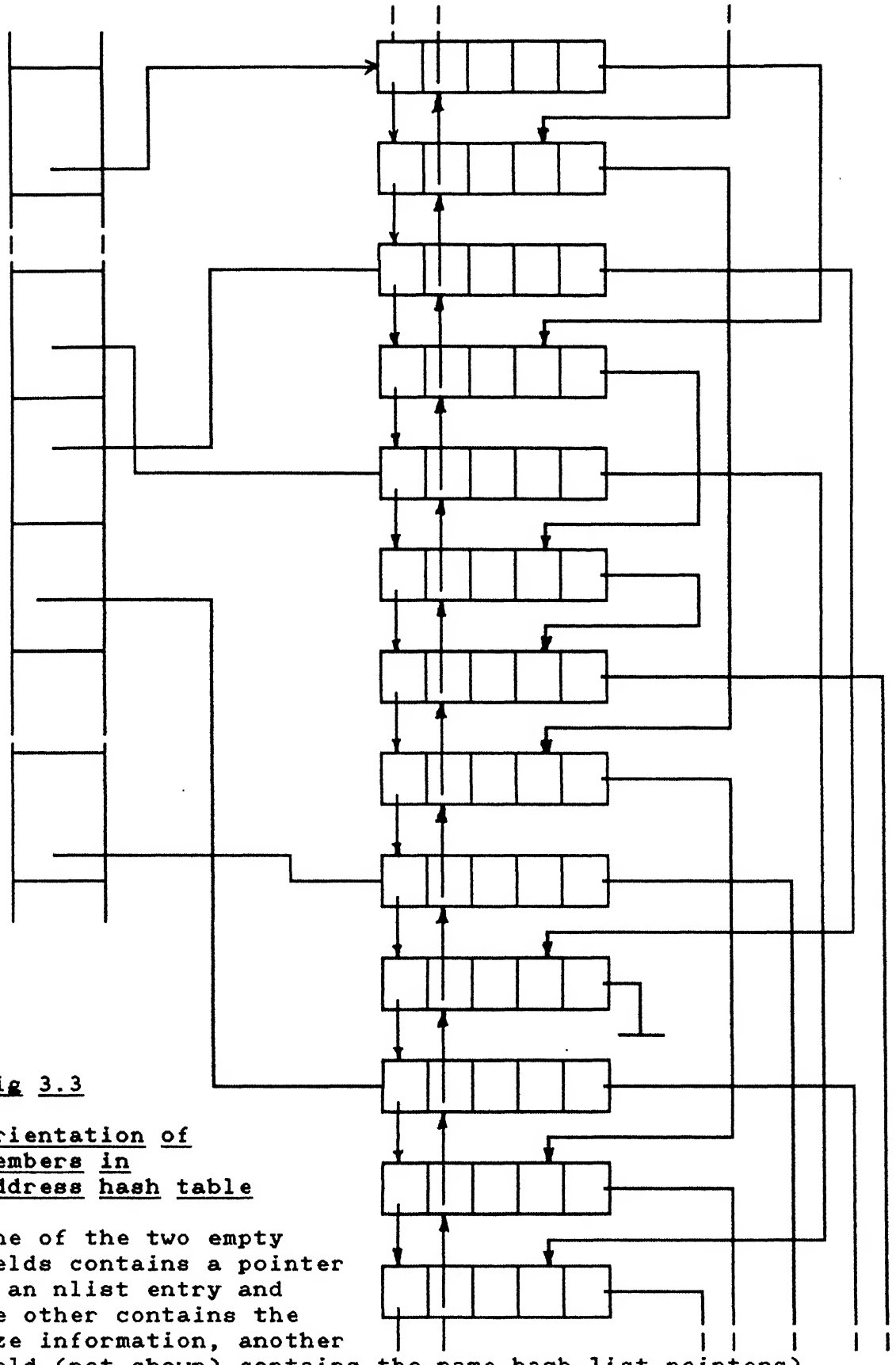


Fig 3.3

Orientation of members in address hash table

(One of the two empty fields contains a pointer to an nlist entry and the other contains the size information, another field (not shown) contains the name hash list pointers)

Algorithm of search for a given address:-

```
search_flag : (INIT_SEARCH, UNSUCCESS_SEARCH, SUCCESS_SEARCH);

ALGO search_address(addr)
begin
      Determine the region (text/data) to which the address (addr) belongs to
      and choose the hashing function accordingly.
      Apply hash function to get index of a bucket in the hash table.
      search_flag := INIT_SEARCH;
      i := index;
      while ( i ≤ maximum index of the hash table) do
      begin
            search_buckets(i , addr);
            if (search_flag = SUCCESS_SEARCH) then  goto ENDLAB;
            if (search_flag = UNSUCCESS_SEARCH) then goto UNSUCCESSLAB;
            i := i + 1;
      endwhile;
      i := index - 1;
      while ( i ≥ minimum index of the hash table) do
      begin
            search_buckets(i , addr);
            if (search_flag = SUCCESS_SEARCH) then  goto ENDLAB;
            if (search_flag = UNSUCCESS_SEARCH) then goto UNSUCCESSLAB;
            i := i - 1;
      endwhile;
UNSUCCESSLAB :
      Declare required nlist not found;
ENDLAB :
endALGO;


PROCEDURE search_bucket(i , addr)
begin
      if (the bucket of the ith entry in the hash table is not empty) then (1)
      begin
            Find out a member of the bucket, so that the nlist pointed to by
            this member has the least positive difference with addr compared
            to the nlist corresponding to any other member in the current
            bucket; (i.e nlist address ≤ addr )

            if (such a member exists) then (2)
            begin
                Set the current member (cmember) to this member;
                if (addr lies in the range covered by the nlist of current
                member) then (3)
                ( i.e nlist address ≤ addr ≤ nlist address + size )
                begin
                    Declare the required nlist to be the nlist pointed by
                    cmember;
```

```
                search_flag := SUCCESS_SEARCH;
                goto  RETLAB;
            end
            else {3}
            begin
                Consider the next member in the same bucket; ( the nlist
                corresponding to it is the entry having least negative
                difference with addr, as nlists are sorted by increasing
                address order within a bucket )

                if (such a member exists) then {4}
                begin
                    Set nlist1 to the nlist pointed to by this member;

                    if ( abs(cmember's nlist address - addr) < abs(nlist1
                    address - addr) ) then {5}
                    begin
                       Set cmember to this member;
                       goto   FOLLOW_BACK_LINK;
                    end
                    else {5}
                       goto  FOLLOW_FORW_LINK;
                    endif; {5}
                end
                else {4}
                    goto  FOLLOW_FORW_LINK;
                endif; {4}
            endif; {3}
        end;
        else  {2}
        begin
            Consider the first member of the bucket;
            Set cmember to this member;
            {the nlist pointed by it must be the nlist with least negative
            difference with addr }
            goto  FOLLOW_BACK_LINK;
        end;
        endif; {2}
    else {1}
        goto RETLAB;
    endif; {1}

FOLLOW_FORW_LINK :
    If (the doubly link list forward pointer from cmember is nil) then
    begin
        search_flag := UNSUCCESS_SEARCH;
        goto RETLAB;
    end
    else
    begin
        Follow the forward link to reach another member of another
        bucket.
```

```
            Set i to the index obtained by hashing the address of the nlist
            that the current member points to.
            search_bucket(i , addr);
            goto  RETLAB;
        end
        endif;

FOLLOW_BACK_LINK :
        If (the doubly link list backward pointer from the cmember is nil) then
        begin
            search_flag := UNSUCCESS_SEARCH;
            goto RETLAB;
        end
        else
        begin
            Follow the backward link to reach another member of another
            bucket.
            Set i to the index obtained by hashing the address of the nlist
            that the current member points to.
            search_bucket(i , addr);
            goto  RETLAB;
        end
        endif;

RETLAB :
    endPROCEDURE;
```

Fig 3.4 Algorithm For Searching the Address Hash Table For a Given Address

pointers in the doubly linked list to find out the desired *nlist* without significant sequential search overhead.

b) HIGH LEVEL

As implied earlier in this subsection, the higher level entries in the global database only imposes C level attributes to the lower level *nlists*. So functions/globals in the higher level always has a reference to the lower level text/data symbolic entries they corresponds to. In KDB any C source file is conceived as a collection of functions whereas information about a function is stored as a linked list of locals declared in its body, a type field signifying the type of returned value of the function (generally returned in d0 register) and two pointers to the two location of a line array signifying the start and end lines of that function. An executable line is signified by an entry in this *line array* where the entry provides the linenumber and the address in the code region wherefrom the set of instructions generated due to this line starts.

In case of globals what we are also interested apart from its *nlists* and relative address allotment in the data region, is the type of the globals. So an identical hash list hashed by name is formed which contains pointers to the corresponding *nlists* but instead of address span of the global or doubly linked list pointers it contains the type information for the global. The way the symbolic information are stored in an executable file has also effected such design superfluousness. This is also demanded from the fact that there are assembly routines and the user may like to avoid debug option during compilation of some files. Notably the later hash list is consulted by the user interface for high level debugging whereas for all lower level address computation and disassembly searches the other hash list is used though in both cases the goal of search is the same set of *nlists*.

However a local variable (auto and register) has no static entity and hence has got no reference to the lower level structure. It is represented like globals and has a type associated with it. But instead of a pointer to the nlist, it keeps information about its relative address with respect to the frame pointer (a6 register in Horizon's cc compiler). In C language the caller of a function pushes the parameters to be passed before the called function saves the earlier frame pointer and redefines its own frame start. Hence the parameters always have positive

offsets whereas the local variables declared in the function body always have negative offsets. In case a local is allocated in a register the register value is stored instead of the offset. A flag in the local variable's representative structure uniquely identifies whether the variable is

- parameter passed in stack
- parameter passed in register
- local variable of the function allocated in stack
- a local variable allocated in a register.

The most complicated consideration, however, arose in the implementation of the type specifiers storage. The difficulty appeared from the fact that we are entirely dependent on the C compiler's symbolic information storage method which generates a group of information sets valid only in the scope of a particular file whereas our aim is a global symmetric database of the type specifiers where a single entry uniquely represents a type over the entire set of files.

For each user specified types (all the 12 basic types are always implicitly specified) the C compiler produces a null terminated character string found in the string table. The string follows the basic syntax:-

<typename>:[t,T]<typenumber★>=<type category><type desc★><other details>

where the typenumber and type descriptor numbers (★-ed) are internally allotted by the C compiler. A descriptor number qualifies more than one type specifier and is kept unique throughout the set of files. On the other hand typenumber allotments though unique inside a file's information set, has no global value.

We represented the whole set of types specified or derived (as described earlier in design issues) by an array (**type_tab**) containing no information whatsoever about the file(s) specifying them. Also in the array each type is uniquly qualified by their typenumber. In our implementation typenumbers stand for the array index of the corresponding type description in the type_tab array. Any type can only be of 7 valid types. Instead of repeated occurrence, the 12 basic types only occur as the first 12 entries (typenumbers 1-12 in the global database) in the type_tab. The 0th entry in type tab is kept unfilled to signify an illegal type.

An entry of the type_tab array contains enough information to unambiguously represent the type it describes. What is stored is based on the following points:

1. the lower subscript of any array declaration need not be stored as C never allows any specific lower subscript unlike PASCAL. By default it is taken as zero in all array declarations. So we store only the higher subscript value.

2. Structures and unions are represented as an entry pointing to a linked list, each unit of which defines its members (or fields) in the same syntactic order as declared (as one travels down the list).

3. Pointer or repeat types contains a typenumber at which they point/repeat. Similar is the case for function types.

For all the above types if they have names that also must be stored for allowing user frieldly queries like *'whatis'* etc. during debugging session. Overall size of any type specifier is also important. Members similarly contain name and size. But the start offset in bits is needed for calculating the actual address of a field in a variable of composite type. Members or fields of a structure/union are nothing but one form of variables as a storage of such composite types can as well be viewed as an array having variable size entries. So the members must also have a typenumber associated with them.

4. In case of bit fields (can be expected in a field of a structure/union), the dummy basic type 12 is used in KDB's database though the C compiler describes them as *unsigned integer* type. A bit field is recognised when it does not start from a byte boundary or does not have a integral size of a byte.

5. An association list is maintained having a set of typenumbers qualified by a descriptor number (C compiler internally keeps it invariant for all similar type specifiers). This list is used to find out any overall repetition of a user defined type specifier.

6. The linearisation of typenumber allotment over the complete set of source files is achieved through the following technique. Except for the first time all the basic types are always skipped. All the later type number definitions and references are incremented by the index value in the type array reached when the current file's first symbolic information is going to be extracted. By

typenumber definition we mean any substring where the typenumber is immediately followed by a '=' character with some definition information following it. All other kinds of occurance of a typenumber will be considered as a reference.

But the implementation of such a global database faced difficulties due to the following properties of the symbolic information generated by the C compiler

i) For declarations in a header file which is included in more than one files, the strings describing the set of declarations in the header file will be repeated in the information set corresponding to those source files. In each case the typenumbers associated with these information (like typenumbers assigned to a type specifier or derived types inside different declarations) will be entirely unrelated.

ii) Apart from the value of the type number, the strings are not also identical in case of different files even when the origin is the same declaration. This may happen when a derived typenumber is introduced within a type specifier's definition string, whereas in the other file's case a typenumber is already allotted to that derived type due to some earlier declaration in the file. Hence only a reference number will be put at the corresponding place in the same type specifier's definition string of latter instead of the full definition as it will be in the first case. For example let us assume one has the following structure specifier found in a header file -

```
struct xxx {
        - - - - -
        - - - - -
        struct yyy  *y_ptr;
        - - - - -
        - - - - -
}
```

and this header file is included in more than one source file. If in one of these files (let file A) and in no other, there exists another declaration -

```
struct  yyy *globl_y_p;
```

and if the above declaration syntactically precedes the header file inclusion statement, the nature of the character string for structure 'xxx' produced due to file A will be different from that due to all the other files. In file A's

information string there will be no substring signifying a typenumber assignment and definition for the derived type *"struct yyy #"* (like substring *''y_ptr:13=*19''*) unlike the case of other files. This happens as the allotment of a typenumber to the derived type "struct yyy *" is already done in the information string for the global declaration (like ''globl_y_p:G17=*32'') and in the substring for the field 'y_ptr' in file A's information set only a typenumber reference number will be put (like ''y_ptr:17'').

iii) Preassignment of typenumbers for a substructure brings in another anomaly. The C compiler does not object if a pointer of an entirely unspecified/unknown structure is used. This produces undefined yet referenced typenumbers (yet to be defined) in an information set. For example if in a set of source files, there is no declaration of *'structure yyy'* altogether then in a substring like ''y_ptr:13=*19'', 13 stands for a pointer to a type specifier having a type number 19 assigned to it in the current subset of information for a file. But as there is no definition for *'struct yyy'* in the definition set of the current file, the typenumber 19 is never defined and hence all such references are dangling.

iv) The method of skipping the character string produced by a declaration which is already analysed in the context of a previously linked file's information set, must also be modified. If such character strings are skipped altogether all the embedded typenumber definitions for the internally derived types will be left unanalysed. This in turn will leave the corresponding slots in type_tab array illegal/undefined. But those embedded derived types may be referenced latter in another description string arising out of another declaration in the same file. For that declaration only a reference to the typenumber allotted to that derived type (in the skipped part) will be made. This reference will be treated as dangling by the global database of KDB though it is not really so. For example let the header file containing the above mentioned structure specification with name 'xxx', be included in two source files (A and B) and let the object file of B be linked later than that of A while forming the executable file. Hence as the string due to the said type specification will be encountered first in the information set due to file A, the association list technique will declare the typenumber due to the string

for 'xxx' specifier as a REPEAT type and skip the entire character string. Thus all the substrings providing internal typenumber assignment and definition like ''y_ptr:13=*19'' will be ignored. Thus any reference to this typenumber later in file B (like ''globl_y_p:G13'' due to global declaration *'struct yyy *globl_y_p'*; encountered at latter part of file B) will become a reference to an undefined type.

This problem was found to occur in case of derived types generated out of pointer declarations.

We enhanced our association list technique of finding repetition in the following way -

Whenever any named type specifier's description string is encountered for the first time, its type index in our global database is entered in the association list entry qualified by its descriptor number. Also the **'depth'** of all the derived pointer type definitions arising inside the current decription string is noted down in an array called **'ptr_off'**. In a composite type specification (like struct/union) there will be a declaration list attached to it, describing the consistuents. By 'depth' we mean the position of any constituent object with respect to the start of the declaration-list. Each entry in ptr_off is actually a pair of depth value and the actual type number (after converting it to our database index) being introduced at that depth. The entries in the association list has two pointers to the ptr_off array signifying start and end of all derived pointer types defined within their description string.

Later whenever a description for a type specifier is found to be a repetition of any previously analysed type (finding a typenumber entry having the same name and type against the same descriptor number in the association list), the range of ptr_off list covered by the previous entry is referred. If in the current description (which is identified as a repetition), an internal typenumber definition for derived pointer types is found at the same depth , the involved typenumber is marked as a repetition of the corresponding typenumber in ptr_off list provided that type entry itself does not signify a pointer type which is dangling (such case may arise due to case(iii) above). In case the earlier entry is found to be dangling, the current definition will be analysed to find out its validity. If in the current definition, the concerned pointer derived type is found to be not dangling, then the earlier entry is made a repeat type of the current entry. This technique thus

solves the problem of dangling type references due to preassignment of typenumbers to the internally derived types.

We would like to mention that DBX is unable to resolve this dangling reference anomaly. This is expected as DBX analyses and maintains the descriptions on a per file basis and it never checks for an undefined pointer or similar type reference outside a file. DBX also makes no effort to avoid duplication of repeated symbolic information. Instead it stores type specifiers as local to the files exactly like the C compiler and thus pays a price in terms of increased initialisation time and storage space. *DBXtool*'s version of *dbx* makes a different effort to reduce the initialisation time by reorientng the symbolic information format in object files. The types are represented as ordered pair of a *file index* and a *type index*. The file index defines which file defined the type and the type index uniquely identifies the type within the defining file. Therefore changing the order in which the header files are included does not change the type indices of the types defined within a given header. If a header file defines ten types, they will always have one to ten type indices and the same file index. During initialisation *dbx* remembers the type information associated with each header file. When *dbx* reads an excluded symbol, it finds the remembered header file entry and maps the header file's type information into a pool of *active* types. Future references to types defined the header file associated with the excluded symbol can now be resolved (real time spent in initialising the debugger is less than 80% of that in original DBX). Details regarding this can be found in [ADA86].

3.4.3.3 **The Disassembler**

Disassembler is an utility that translates machine language instructions into assembly language form. As the design and implementation of the disassembler is completely CPU family specific, we shall not discuss the details about it. A disassembler should take care of the following facts -

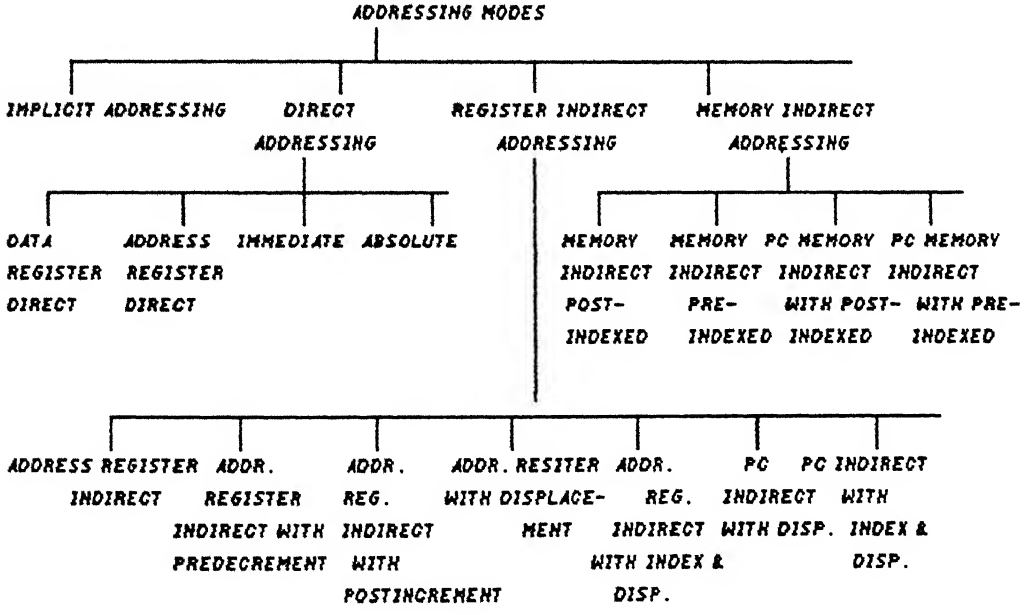
1. The assembly language produced should conform to the conventional assembly language used for the host CPU's family (preferably in the same syntax of assembly that the assembler residing in the host m/c accepts). In our design we followed the standard assembly convention for MC68K series (unlike *adb*). [LEV87] [MOT85]

2. All the absolute and global names (may even arise from high level code) should be retained as it is interpreted in the assembly. For our case all the names defined in the assembly code part of IITKIX code is kept and stored as it is whereas any C function name will be preceeded by an underscore in its assembly representation (as done by the C compiler). However in data segment symbols the names are kept as it is declared in C.

3. All user specified labels should be preserved and reproduced. However we made no such effort as in the disassembly produced, each assembly statement is always preceeded by a *<routine name>+#<offset>:* label which serves almost the same purpose as references are also produced in the same form.

4. The various addressing modes of the host CPU must also be properly represented in the disassembled code produced.

KDB's disassembler interpretes the address modes in MC68020 which can be best understood by the tree in Fig 3.4.



We have taken care of all addressing modes except the memory indirect addressing as the current C compiler never produces a code using this addressing mode. In

case of absolute long word addressing we use the search technique described earlier to produce an assembly containing the symbolic representation for any address (if any). The m/c level coding done by the assembler from the assembly can be best understood from the instruction format of MC68020 for different instruction types [LEV87], [CRA86] & [MOT85]. The disassembler immitates the encoding process in the reverse direction.

3.4.3.4 The User Interface

The two major parts of the user interface is the simple linewise editing facility for the user to erase the previously typed character(s) (using backspace key) and the display of debugger's reply to a query made by the user. We shall only discuss the technique adopted for displaying C source lines and printing the value of a variable (or expression).

As already mentioned, the console is used as a serial port. The routine *'printf'* is enhanced and a C library *'scanf'* like function is developed using the lower level routine *'siogetchar'* which waits in a busy loop for a character to be received from the console control port. Our version of scanf accumulates upto 100 characters typed in a line without being erased until a carriage return is encountered. Our scanf always consumes all the characters typed in a line even if it does not partly or completely match the format string of the scanf call (unlike C library scanf). A character erasing is displayed for user's convenience by printing a backspace followed by a space and finally again a backspace. As a direct corollary of printf and scanf, *'sprintf'* and *'sscanf'* functions are also developed. The *'scanf'* and *'sscanf'* return the successfully matched and assigned input terms.

For displaying C source files a completely different filesystem is used. The filesystem format is kept very simple (a contiguous file placement with an initial gap of a few blocks between two succesive files) and it supports no subdirectory structure. The file system header (always a predetermined number of blocks at the begining of the disk partition containing the filesystem) is a collection of ***filenames of the constituent files*** and their ***start block numbers*** along with the ***span of the file information in terms of blocks***. A marker is used to explicitly signify the end of file header's information set. The *eof* character used for the text files stored in the file system is chosen to be (char)1. More detailed information about the actual filesystem can be found in the user manual for KDB (see Appendix B).

KDB keeps its own set of buffers and uses only low level block device i/o routines for retrieving blocks of data from the filesystem. The buffers are managed by a set of header attached with each of them which keeps information like the disk blocknumber of the data, the buffer is currently holding; a field keeping track of the reference history of a buffer etc. The block replacement policy followed is LRU. This choice follows from the fact that the users tend to concentrate on a particular region of code and the block size (512 bytes) is sufficient enough to contain the locality of reference. This aids KDB to aviod disk i/o during a debugging session. However the same set of buffers are used during the initialisation phase and then a FIFO replacement policy is imposed as the nature of information processing at that phase conforms to this pattern.

To reduce the overhead the overhead of doing a character-by-character search looking for a line feed, to determine the source lines, KDB preinitialises the offset (in terms of bytes) of all executable lines of C code from the beginning of the filesystem. This preinitialisation is done during KDB's initialisation phase. The *line array* introduced in subsection 3.4.3.2.2 contains this information along with the pair of line number and its corresponding code address. Finding out the offset of a particular executable line in the initialisation was justified as -

- it will reduce the overhead of searching for every query.
- during initialisation the code is analysed sequentially, the line offset searching conforms to this pattern and hence overhead is less.
- the information can be read out from disk in larger chunks during initialisation as the whole code will be analysed sequentially.

This added task during initialisation has no doubt slowed down the process as it involves disk i/o which is inherently a slow process, but this overhead is found to be useful for the later phase. Now in the interactive phase KDB knows the offset of a line directly, if it is executable and in case of a nonexecutable (like a comment line or a blank line) line the overhead of search begins only from the last executable line preceeding that line. If no such line is found (like header inclusion statement lines at the very begining of the file), the search begins from the start block of the file.

Displaying the value of a variable location is one of the most desirable features of the debugger and this feature should be intellegent enough to print the

information in proper format complying to the user's expectation. The convention is followed and C syntax expressions along with absolute addresses a recognised. A simple parser interpretes the C syntax and the KDB symbol databa is consulted for computing the address and size of the memory to be displayed. format is decided by checking the type of the symbol. For printing globals address from the symbol table can be directly used whereas for parameters offset found in the symbolic entry is to be added to the current frame pointe value and for locals the offset is to substracted. This follows from the technic adopted for implementing function calls in C language. The value of the regist symbols are found in the registers, where the corresponding symbolic entri provides the register numbers. The manual provides details of various commar and their syntax.

3.4 CHANGES REQUIRED IN KERNEL CODE FOR SUCCESSFUL OPERATION OF KDB

The kernel services needed by kdb caused a set of minor changes in t kernel code . We mention them to justfy the changes.

a) When block i/o is invoked by any process it goes to sleep (unless it is asynchronous i/o) waiting for the event of i/o completion. This sleep calls for context switch. The debugger cannot afford such context switches in the interact phase as it is not a part of normal processing done by the kernel. So it waits i busy loop setting the processor at a sufficiently low priority allowing the d interrupts to come in. The disk interrupt service routine in normal case chec for completion for an issued i/o, if the i/o is not totally completed it reissues i/o call with new set of parameters or else sets up the i/o completion flag in buffer designated for that i/o and wakes up the processes sleeping for the ca of the i/o completion in that buffer. For KDB's i/o no such step is required as t will ultimately change the orientation or other linked list and the u-parameters. a condition is incorporated in the *'iodone'* routine to skip this portion in case debugger's i/o.

b) As mentioned earlier the clock service is skipped when the debugger g control.

c) The trap service routine sets the pointer of user register block (u_ar0) to

region in the stack where the user register values are stored when an exception occurs. In case of debugger's exception processing this parameter should not be changed as that will leave this pointer dangling when the debugger's exception processing is exited. A small condition introduced in *'trap'* routine takes care of this fact.

d) In the *'vmstart'* routine a change is made to allow overwriting in the code segment.

e) All the parts of the code involving an explicit setting of the status register is modified to restore the value of the trace bit found before modification.

3.5 PORTABILITY ISSUES

This debugger is concerned with a Unix like operating system and it is quite expected that IITKIX on its phases of development will be ported to other m/cs. So portability issues are kept in mind during KDB design.

1. Nowadays almost all high end microprocessors allow the facility of m/c instruction wise stepping. So whatever the CPU may be (even Intel's range of μPs) we can expect a trace bit in the status register. MC68020 allows more selective tracing by having two trace bits (hence two different form of tracing) in the status register. We could have reduced the overhead of halting at each intermediate m/c instructions encountered during line stepping by using the form of stepping which generates a trace trap only on a change in program flow etc. (e.g if during linestepping a function call is made, we could skip stepping until a return occurs). But taking into account the portability factors we avoided it.

2. A CPU being used by an Unix like O/S must provide at least one trap like instruction for implementing system calls. A software flag (in a register) can easily distinguish a system call from a KDB break even when there exists only one such way of invoking kernel services by a user program (i.e only one such trap instruction). MC68K series provides a *BKPT* instruction which can be used to generate an *illegal instruction trap* on execution. We avoided use of it for implementing breakpoint as it is highly Motorola design philosophy specific.

For similar reasons the set of illegal and unimplemented instructions (they also generate illegal instruction trap on execution) were not considered.

3. KDB during interactive phase is completely independent of the filesystem structure of the O/S kernel it is debugging. So it can be easily made a universal kernel debugger. The private filesystem adds to the portability of KDB. It can be made completely independent of the kernel's filesystem structure by keeping the executable file of the kernel in its private filesystem instead of keeping it at the root of O/S's filesystem. This will also make KDB completely independent of all higher end kernel service functions.

4. For displaying and receiving user's command the two lower level functions *'siogetchar'* and *'sioputchar'* are to be modified when IITKIX is ported to another machine.

5. The disassembler part of KDB is to be completely rewritten if the new host m/c's CPU is not MC68020 or the lower range CPUs in the same family.

6. The code for symbolic database formation is entirely dependent on the symbol table format in an executable file. In case the compiler being used for IITKIX development produces a different symbol description format, this part of the code must be modified.

3.6 THE DEVELOPMENT PROCESS OF KDB

As KDB should be developed to run in kernel mode, like all prior implementors we also were quite careful in designing and testing of KDB's code. Added care was required as the kernel code was not made write-protected (as was done previously) as per requirement of operation of KDB. In fact the development phase can be subdivided into following two stages -

i) developing a user level debugger using the same philosophy of operation as that of KDB.

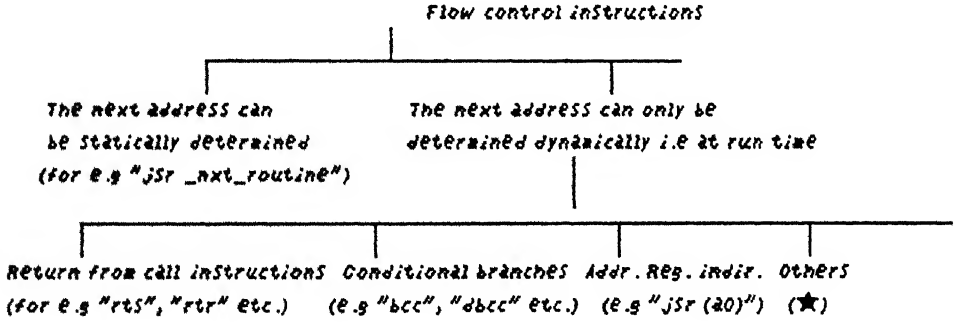
ii) bootstrapping KDB at the phase of actual testing.

At the initial phase we developed the disassembler and the symbol table information

extractor completely at the user level. However the testing of break/step sequence was harder because of the fact that while running a user process one cannot set the trace bit in status register as the corresponding instruction is a privileged one and only a kernel process (or a process running in supervisory mode) can set it ON. Further the DEBUGtrap instruction was of no use for implementing breakpoint in this case as the trap is serviced by an already residing (4.2Bsd Unix) kernel. In our user level version the breakpoint was implemented through an *Illegal Instruction (opcode = 0x4AFC)* which when executed, the kernel produces a Unix *'signal'* called SIGILL. We place the break/step manager as the signal catcher function.

In this debugger the concept of next instruction to be executed is very important as the stepping is achieved by treating the next location to halt as a temporary breakpoint i.e if the address of the next m/c instruction to be executed can be found out (may be statically or from run time values) then the instruction at that address is modified to an illegal instruction as done in case of breaks. We briefly discuss the method adopted for determination of the address of the next instruction to be executed.

We classified the set of brach instructions on this basis in the following way



In case the current break (may be temporary) is at a normal instruction causing no change in program flow, the next instruction to be executed is always the instruction that follows it in memory. In case of a flow control instruction of first category (in the above tree) the next instruction address can be determined by scanning the instruction operand (in locations of user specified breaks they are needed to be computed only once). But complication arises in case of the other categories. In conditional branch instructions the break/step manager effectively computes the condition (after consulting the saved value of condition code register) to determine whether the next instruction or the branch instruction will be

executed. In case of *return*-s the stack and in case of register indirect the corresponding register (the register number can be found while disassembling) is consulted for determining the next instruction's address. The starred(★) kind takes care of all other possible modes of specifying the branch address and is not taken care of as it demands numerous cases to be taken into consideration but fortunately the compiler seldom uses these modes.

We faced another difficulty due to the fact that when signal catcher function gets control a kernel generated information set is found over the stack built upto the point of occurence of the illegal instruction (NB:- Unlike other debuggers we have only one process where the code being tested resides with the debugger's code in the same code region and hence they use the same stack). The exact format of that information added in the stack was not known to us and moreover that part of stack contains vital information like the saved value of program counter and the saved value of condition code registers along with other hardware registers when the illegal instruction was being executed i.e at the time of break. These information are to be retrieved properly for successful operation of the debugger. So we used intuition to recognise the required places by using a hit and trial method and by scanning this part of stacked information with the help of *adb*. Another point worth mentioning in this context is that we could not use any high level Unix debuggers for checking this user level version of break/step manager also, as all signals are caught by the parent debugger process which then does not proceed any further. The bootstrapping phase is quite self-explanatory and is not discussed in any detail in this thesis. The way KDB is to be installed, invoked while booting and can be used is discussed in the user manual for KDB.

CHAPTER – 4

CONCLUSION

KDB is built, tested and is now running successfully with IITKIX kernel. The way the debugger's power can be extracted entirely depends on the developer's own approach. In general KDB demands some minimum hardware and software supports for its succesful operation.

4.1 THE MINIMUM SOFTWARE AND HARDWARE SUPPORT REQUIRED FOR KDB

1. A trace bit for halting m/c instruction wise.
2. A trapping facility.
3. A disk partition for its own filesystem and an utility in the developing environment for building up and updating the filesystem (we developed this utility in our environment).
4. A facility of transferring the executable file of kernel from the filesystem where the operating system is being compiled to the filesystem of the IITKIX O/S. This utility is also developed in IITKIX's environment.
5. Enough memory for building up KDB's moderately big database with sufficient space left for other user contexts.

KDB can easily be used in debugging any other version of UNIX alike O/S's kernel running on a MC68K family, if the source code for the part tackling interrupt servicing is available. Installation of KDB as the signal handler for the DEBUGtrap in that kernel's exception handler vector table and a function call to KDB's initialising routines from kernel's main will ensure succesful working of KDB.

4.2 LAPSES OF KDB AND SOME TIPS TO BYPASS THEM

a) KDB is completely sensitive to virtual memory management in the sense that it cannot tackle the physical memory by itself. As the kernel faces a breakpoint, the page of U-area is used as the supervisory stack. So when this entry of page table is made invalid by kernel's virtual memory management code for mapping in another process's U-area page during context switching, any break or step request at those

region of kernel code can cause a complete malfunctioning. This can be avoided by setting a break-point at the point in the kernel code where the process (under the context of which the break is faced) is expected to return after being woken up and continuing from the point of halting without involving step of any kind. Currently as explained earlier an automatic break_point is introduced at the next line when a C function call eventually leads to a context switch.

b) The trace bit in the status register is used by KDB (note that *'ptrace'* utility also uses the same trace bit but in its case the trace trap comes from the user level process and hence get serviced by a separate handler by checking whether the supervisory bit is ON or not). So if any statement of the kernel code explicitly modifies the status register and if it directly affects the trace bit (for e.g a command like *'movw 0x2700, sr '*) then the debugger may loose control over the kernel's execution. The kernel is interested in the status register mostly for setting the processor at different priority levels. Any implementor should take care to save the value of trace bit before such modification and finally *'or'* in the value after the operation. The routines settiing processor priority levels (*'spl'*) along with the routines for involving such statements have been modified to take care of this problem.

c) KDB uses a trap service routine written in C (*'trap'*) which is common for all other trap services done by the kernel. So any break in that routine will cause an infinite loop of break trap exception and exception service ulimately overflowing the supervisory stack to modify the U-area and even the code segments causing irretrievable damage to the host m/c's configuration.

d) KDB uses the block device i/o strategy function (*'sdstrategy'* in IITKIX) and hence all functions used by it. Also the device interrupt service routine (*'SDintr'*) is involved as it determines whether the i/o initiated is completed or not and if found complete, raise some flag to indicate that to the waiting process. So a break or flaw in those routines will again render KDB service ineffective.

e) Breaks at hardware clock service routines should be avoided as they are executed very frequently and may cause a series of break exception before KDB has completed servicing the previous break.

f) Real time nature of the O/S services can never be restored while inspecting the code with the help of such a debugger. When the debugger is involved the process is much slower as after each m/c level instruction an exception processing is involved. For example if one line_steps through a higher level routine of block device i/o, it may be found that the involved process will not even go into sleep for a single time as by the time the statement is reached, the i/o is already over. A proper scenario can only be judged by designing the test case very carefully and by chosing proper set checkpoints accordingly to break in, while continuing with the kernel execution in the intermediate regions.

4.3 ENHANCEMENTS AND EXTENTIONS SUGGESTED FOR KDB

Some enhancements can be made in KDB without much difficulty. But one issue must be kept in mind while doing any such effort. KDB while interacting uses the supervisory stack which happens to grow from the end of the page containing the U-area and any of the U-parameters should not be disturbed during KDB operation. Further if the stack grows and overwrites the U-structures data, kernel will start behaving in a completely strange way. So glut of local variables and recursive routines should be avoided. The enhancements suggested are as follows :-

1. KDB can be easily converted into a **performance analyser** by making provisions for counting how many times a function is called or a process wakes up and sleeps again on a particular event or due to nonavailability of a particular resource etc.; and introducing similar performance evaluator flags to very important global variables. A minor modification of symbol table database can easily achieve these goals. However the real time behavior can never be guranteed due to the fact depicted in the earlier section. One can refer to a M.Tech thesis from IIT Kanpur [BHA87] for devising more sophisticated method of performance analysis with the help of KDB.

2. To make KDB more user friendly commands like scanning down the stack for finding all activated instances of a local variable or provision for defining debugger's private variables etc. can be introduced.

3. A much expected feature of a debugger is the conditional breakpointing and

conditional tracing. A condition is usually specified with a break point in C language syntax and the break is only executed if that condition evaluates to to nonzero value. A conditional tracepoint produces a message on the screen whenever the specified conition is satisfied. This can be incorporated as a useful enhancement.

4. In kernel operation, an optimised code is very important for getting improved performance. But the currently used C compiler does not optimise with the debugging set. If at any later phase a more developed C compiler is used for IITKIX development, KDB's displaying technique has to modified and a more cunning break/step control may be required.

5. The debugger can be made completely independant of virtual memory by having a context reserved for it permanently and setting the kernel to that context whenever it gets control. It also should have a private place to copy the page tables and thereby imitating the kernel's virtual to physical address translation procedure whenever required. But this overhead can only be justified if enough contexts are left with the debugger residant to accommodate in the user processes. In an architecture like HPPA (Hewlerd Packard Precision Architecture) this concept can be justifiable due to high number of contexts in a very huge virtual memory.

4.4 MULTIPROCESSING ENVIRONMENT AND KDB

KDB's principle can even cope with a kernel in multiprocessing environment which uses master slave configuration. In a completely parallel environment where a minikernel resides in all processors, a completely different approach may be necessary.

We hope KDB will have a significant impact on the pace of later phases of development of IITKIX and will relieve the implementors a great deal from the pain and agony for getting their codes working.

--- || ---

APPENDIX – A

EXECUTABLE FILE FORMAT IN UNIX
AND DETAILS OF SYMBOLIC INFORMATION FORMAT

A.1 THE EXECUTABLE FILE FORMAT IN UNIX

Though different versions of Unix differs in the executable file format so far as the minor details are concerned, overall format is the same in all releases. We shall present the format generated by a high level language compiler with the debugging option set. If the option is absent the symbolic information part will be short. An executable file has the following sections -

1. A *Header*
2. The program *Text*
3. The *Data* area
4. *Relocation* information
5. *Symbol* table
6. A *String* table

The above mentioned sections occur in the executable file in the same above-stated order. The last three sections will be omitted if the *strip* (*-s*) option of *linker(ld)* is set.

Currently IITKIX source files reside in a BSD4.2 filesystem and hence the C compiler used to compile them and produce the executable module of the kernel is the cc *compiler* provided by the vendor. The manuals and the /usr/include help files gives a most of the details of the file format produced by the compiler. But for completeness sake some major points are repeated here.

The Header : The header can be best described by a C structure **exec** which in fact is used by the exec routine (and the bootstrap loader in case of IITKIX) to load the executable file in memory. The structure is

```
struct exec {
   long a_magic;/*magic number*/
   unsigned a_text;/*size of text segment*/
   unsigned a_data;/*size of initialised data  segment*/
   unsigned a_bss;/*size of uninitialised data*/
   unsigned a_syms;/*size of symbol table*/
   unsigned a_entry;/*entry point*/
   unsigned a_trsize;/*size of text and*/
   unsigned a_drsize;/*data relocation*/
}
```

THE TEXT and DATA Regions : As is well known in Unix alike systems the memory image of an executable code i.e the process has three regions; **text, data** and **stack**. The 'exec' routine loads the text region from the executable file as it is; whereas in data region the initialised part of it (containing gloals, character strings and places for returning structures in case any function does it) is loaded from executable file. There is no space left in executable file for the *bss* or the uninitialised data area, so in core representation of it is created by skipping the amount of bss size mentioned in the headerand initialising this part to zeroes. A segment is then attached for the stack area putting it at the highest possible location in the virtual memory for core image. Stack grows downwards from virtual addess **0x7ffffe** for the *MC68010* with VME bus system and from **0xffffffe** for *MC68020* with VME bus system (as it is with current system). In case of IITKIX the idea of kernel stack (or supervisory stack) is somewhat different. As kernel sets up the page tables for itself, it sets the begining of the stack at the highest address of the page to be mapped to U-structure and the stack grows downwards as usual.

The Entry Point : The entry point is another very significant information which is used by the 'exec' routine or the bootstrap loader to set up program counter for the process, so that the code loaded at that address is executed first.

No loader loads the relocation, symbol and the string table in the memory unless they are "cheated" by modifying the initialised data size in the header information in the executable file.

The Magic Number : It actually determines the type of the core image produced. Firstly it is checked by the 'exec' routine and only one of three valid number signifies that the file is executable. The loader refuses to load it if it finds an invalid number. The three valid numbers are octal **0417 (OMAGIC) , 0410 (NMAGIC)** and **0413 (ZMAGIC)**. Header is not loaded in OMAGIC files and in core image of such executable files are not made write-protected and shared. So self-code modification is possible for only such files (which can be produced with *-N* option during compilation and linking). For such files the data segment is immediately contiguous to the text segments.

In case of the other two kinds of executable files (NMAGIC and ZMAGIC) the data region begins at the next page after the last page in text region and the text

region is write protected.

The Symbol Table : The constituents of the symbol table are a number of symbolic structure units called ***nlists***. The number of such entries is almost directly proportional to the size and declarations in the source code. Presence of this symbolic informations enables, the high level language interpretation capability of a debugger. Details about the structure of an entry (*nlist*) can be found in the manual for *a.out* and in *stab.h* of /usr/include. We repeat the basic structure for completeness sake.

```
struct nlist {
    union {
        char *n_name;/*for use when in core*/
        long n_strx;/*index into file string table*/
    }
    unsigned char n_type;/*type flag*/
    char n_other;/*not used*/
    short n_desc;/*type descriptor*/
    unsigned n_value; /*value of this symbol*/
}
```

The actual name of the symbols are kept in the string table and the *n_un.n_strx* field gives an index into the string table. The *n_name* pointer is kept for convenience of the debuggers which actually reads in the informations from the string table region of the executable file and makes the pointer point to those locations. The *n_type* field gives the type of a particular symbolic information. The symbol table entries can be of text, data, absolute data etc. informations or can informations about local, parameter, register, global, static etc. type variables. The debuggers processes the *n_name* feld according ly. The descriptor number can subcategorise a group of symbolic informations. The *n_value* contains the address information for any symbolic entry in most of the cases.

A detailed description of the symbolic information and the proper way of interpreting it is provided in the next section as the manuals available do not provide proper details in this regard.

Figure A.1 pictorially depicts the organisation of a typical executable file in Unix-like systems.

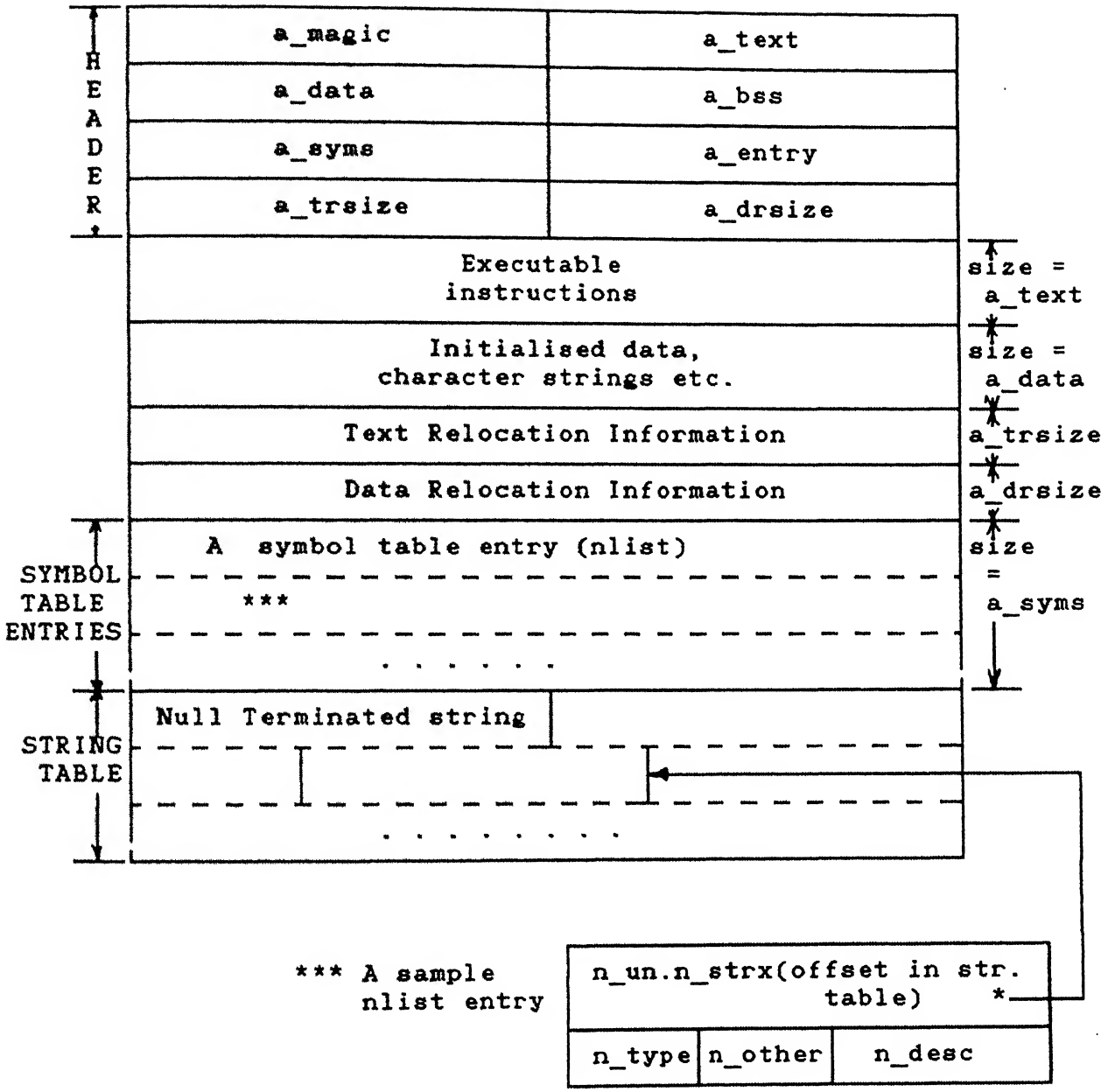


Fig A.1: The Executable File Format In Unix

A.2 DETAILS OF SYMBOLIC DESCRIPTION FOUND IN *nlist-S*

All the symbolic information is available in the form of *nlists*. For any type specifier, typedef or variable declaration a symbolic information entry will be produced where the corresponding character string found from the string table contains a description of that declaration in encoded format. A typical string contains typenumber definitions and references where typenumbers appears in decimal format. In the subset of a symbolic informations for a particular source file, the typenumbers uniquely describe a typespecifier (internal or external) unless the reference is dangling.

Apart from these a symbolic information for each executable line and the informations due to internally generated labels also appears in the subset. The informations about left and right '{' brackets with their nesting level are also available. The *n_type* field in n_list information uniquely describes the nature of any entry. Table A.1 gives some tips for interpreting the informations.

In case of structure/union specifications, the structure declarator list is explained in "other details" (see table A.1) part. Table A.2 describes the syntax of the substrings defining typenumbers internally generated by the compiler or assigned to an user specified type.

For the sake of presentation, let us assume that a properly developed C source file follows the organisation provided below.

1. A set of header file inclusion statements The header files contains all the necessary type specifers and global declarations.
2. A set of function declarations along with function bodies. (NB:- we are not concerned with the symbols arising out of *'extern'* definitions.)

Then in the subset of informations due to that file will contain the symbolic informations in the following order.

1. Source file decription (N_SO)
2. Object file description
3. 12 basic types (with *integer, char, long, short, unsigned char, unsigned short, unsigned long, unsigned int, float, double, void* always been allotted typenumbers 1 to 12 respectively).
4. a) i. First include file's decription
   ii. strings describing type specifiers and global declarations in the

include file.

b) .....

- - - - - - - -

n) Descriptions for last (nth) include file.

5. a) i. description of first function definition in the file

ii. description for any local type specifier

iii. description strings for register variables if any.

iv. description strings for parameters

v. description for locals of the current function

vi. a set of entries describing all the executable lines in current function

- - - - - - - -

m) Description for last (mth) function definition in current file.

6. code and data segment symbols (code symbol names are functions declared preceeded by an underscore; data segment symbol names consists of the absolute symbols (if used in this file) and global or static variable's name preceeded by an underscore. NB:- This part of information is only available for a file if the debug option is not set during compilation).

Fig A.2 describes the orientation of symbolic database for type specifiers produced by KDB's symbol table information extractor. As an example two instances of the part formed due to the structure specifier 'xxx' described in table A.2 is shown when this specifier appears in two different files. Note that in the second instance, explicit storage for the member descriptions is avoided.

TABLE A.1: nlist TYPES AND CONTENTS OF OTHER FIELDS

n_type	Char. signif-ing type	n_name	n_value	n_desc	Actual meaning
N_GSYM	G	<globl_name>:G<type no><optional definition>	Abs. addr of the variable	Descript-or number for the typeno.	Global variable
N_STSYM	S	<static_var_name>:S<typeno><opt. defn>	-do-	-do-	Static variable
N_PSYM	p	<param_name>:p<type no><optional defn.>	+ve offset in bytes wrt fp(**)	-do-	parameter of a function
N_RSYM	r	<reg_sym_name>:r<type no><opt. defn.>	number of register	-do-	register variable
N_LSYM	t	<typedef_name>:t<type no>=<defn. explaining the desc.>	"	"	typedef declaration
	T	<struct/union_name>:(s|u)<desc. num.><substr. describing the members>	"	"	struct/union type specifier
	(0-9)+	<local_sym_name>:<typeno><opt. defn>	-ve offset wrt fp(**)	desc. no. for type	local(auto)variable
N_FUN	F	<func_name>:F<type-no><opt. defn.>	actual text addr. from which its code starts	-do-	function definition
N_SO	"	name of source file as in development environment	first executable addr.	"	C source file name
N_SLINE	"	"	code region addr. of the start of the instruction set got by translating the line	line number	executable line's information

** fp = Frame pointer

Table A.2 Format of Definition substrings for internally Derived Types

TYPE	SYNTAX OF DEFINITION SUBSTRING	EXAMPLE
Basic type	=r<desc.>;<lowest val> ;<highest val>	For integer type:- "int:t1=r1;-2147483647;2147483647
Pointer type (P)	=*<typenumber>	For a function definition like 'char *func_nm(- - -)' :- "func_nm:F13=*2"
Array type(A)	=ar1;0;<highest valid array subscript>;	For a declaration like 'int arr[17]' "arr:G23=ar1;0;16;1"
Function type (F)	=f<typename>	For a declaration like 'typedef unsigned short FUS();' :- "FUS:t14=f6"
Structure type (S)	=s<desc.><description of member#1><desc. of mem#2>...<desc. mem#n> where member description substring has the following syntax: <member name>:<member typenumber with optional defn>,<start offset in bits>,<size in bits>;	For a declaration like 'struct xxx { int *memA; char memB[10][20]; union { int (* memCA)(); struct xxx *memCB; } memC; }; ' :- "xxx:T21=s108memA:13,0,32;memB:23=ar1;0;9;24=ar1;0;19;2,32,200;memC:25=u201memCA:27=*28=f1,8,32;memCB:29=*21,0,32;;,200,32;;"
union (U)	=u<rest same as structure specifier>	

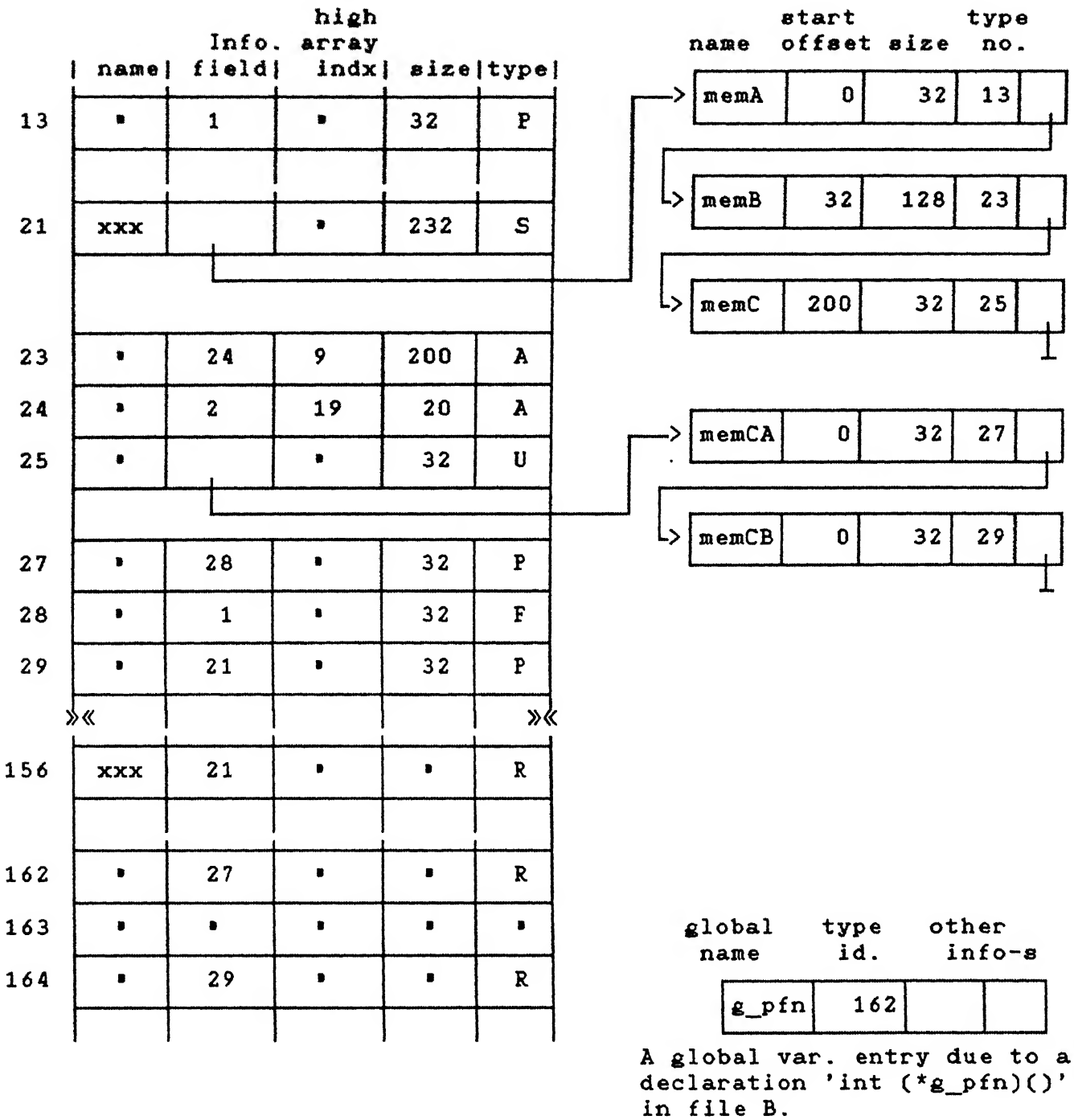


FIG A.2: THE TYPE SPECIFIER DATABASE

(Here the lower valued type indexes stands for declarations in File A whereas the higher values stands for declarations in File B. A global variable entry having a reference to the relevent part of the database is also shown)

APPENDIX – B

A USER GUIDE TO KDB,
THE KERNEL LEVEL DEBUGGER FOR THE IITKIX

SECTION I
INTRODUCTION

IITKIX is a uniprocessor operating system being developed in IIT Kanpur for last few years. The implementers have quite often felt the need for a debugging facility for verification and testing their o/s code. This debugger essentially meets that need and will aid the future implementations and extention of this operating system. The kernel debugger as will be addressed as KDB for the rest of the manual is capable of operating in directly assembly as well as C source code level. Here we present a brief discussion on the the working principle and environment needed for the debugger. The principle characteristics of the debugger are as follows :-

a) So far debugging in the assembly level is concerned the debugger conforms to the addressing conventions of adb. KDB has got an built-in disassembler which produces the assembly code back from the translated code provided a correct starting address is mentioned to it. The disassembler is equipped with instruction set of 68020 processor along with all the downward versions. Assembly language conventions followed in the disassembly is standard assembly language convention of 68k series [REF 1] (unlike adb).

b) KDB is almost exactly like DBX so far as source level debugging is concerned with some added utilities. The later discussions and especially the section on commands (Section 3) gives a clear picture how the and where KDB differs from DBX.

c) KDB unlike other user level debugger is not compiled seperately and kept as an utility. It is essentially a part of the kernel code and after initialisation gets control only through a special trap instruction (Trap 2 is used currenly). So KDB is a set files which is to be compiled with the other kernel source/object files (The way it is to be done is described in Section 2). One thing must be kept in mind while doing this linking is that the kernel "main" function ,after initialising all the system resources and software resources, should make a call to the setup function (namely start_kdb() now) for the KDB to be operative.

d) KDB runs in the environment of the process that encounters the breaks and steps on request. As that process is the running kernel of the o/s so it always resides at the same virtual address space as process 0. This

feature is unique as almost all other user level debuggers spawns a child process on which it overlays the environment of the code to be debugged and uses system calls(like ptrace in UNIX) to explicitly control the childs execution. Clearly that procedure cannot be adopted to KDB. Actually KDB works on the principle of code modification on the code to be debugged. At user specified break point addresses the code is replaced by a trap instruction which actually gives the debugger control back on a break. Stepping is essentially achieved through setting and unsetting of the trace bit(t1) of the status register.

e) Another unique feature of KDB is that it uses a seperate file system for reading the C source files. This is because of the fact that

>> IITKIX file system is a SYSTEM III filesystem and IITKIX is being developed nowadays in BSD4.2 filesytem. Hence all the its source files lie in the later filesystem. It is quite time consuming and annoying to transfer the files from one filesystem to another. Above all if source files are also kept in the IITKIX filesystem , this can produce space limitation problem on the disk partition containing IITKIX filsystem (currently /dev/sd2a) in future.

>> Normal kernel read write routines cannot be directly called by the debugger because that can disrupt the current buffer list orientation, open file tables, blocked processes' list and many other kernel data structures which itself may be of current debugging interest. Also this will lead to a difference between the performace of the o/s code with and without the debugger, which of course is highly unacceptable. So the KDB maintains its own private buffers and avoids invokation of alien kernel routines as far as possible. A private simple file system also helps in keeping the i/o handling part of the debugger code simple.

>> Morover a private filesystem structure increases the portability of KDB.

THE PRIVATE FILESYSTEM OF KDB :-

A description of the file system format is presented below. (NB :- a block means contiguous 512 bytes on disk)

1. A Header of 4 block size -

The format of the header is described by the structure ***f_info_filsys*** in a help file named *"kdbfilsys.h"* in $BASE area. In the header there is an entry for each file in the file system, where the first 35 bytes signifies the filename,the next 4 bytes gives the block number where from the content of the file starts and the last 4 bytes the size of the file in blocks, i.e the file can occupy this amount of contiguous blocks at the most(see later for explanation). The end of the file information in the header is denoted by a special entry in the Header *f_info_filsys* list which has a name "**$$$**".

2. The data area -

This can be of any size (of course not exceeding the maximum no. of blocks in the partition where the file system is created). The content is essentially text characters of the files the filesystem is made of. The **end of a file** is denoted by **character 001**(eof in this filesystem). When the filesystem is first created, in the area containing the contents of a file, 2 blocks are left after the last block (containing the eof) for further expansion of the file ,as user may like to modify it in the process of debugging his code. The initially empty 2 blocks tend to fill up as one file grows in size (see later). The eof character is also moved towards the end of the set of contiguous area allocated to that particular file.

f) There are a few C compiler anomalies and KDB reacts with those just like DBX does: The anomaly may arise because

  )) One has declared a composite datatype (structure/union) having a pointer reference to undefined structure or union. For e.g a declaration

```
struct xxx {
        ....
        struct unknown *ptr;
        ....
};
```

will create no compilation error in the whole translation phase even when the structure "unknown" is never defined anywhere in the whole set of source code. The debugger treats such cases as "(Class 0)" just like DBX and any refenece to this field will be "UNPRINTABLE".

)) This can also occur in case of misarranged include files. For e.g if file "a.h" contains the above mentioned definition of struct "xxx" and file "b.h" contains the definition of structure "unknown" but in no C source file both of them are included together, then the same paradox will take place.

)) In the current HCL HORIZON C compiler interger and long is treated identically as they both signify a 32 bit storage space. So any refence to long is actually a reference to integer . So KDB as it gets these informations only from the compiled executable code, will also reflect the paradox. Any variable of long type (or unsigned long) in source code will be of integer (or unsigned integer) type to the debugger. But in a different compiler (in a different m/c) which really treats these two basic types diffently, this anomaly will not arise.

g) KDB like other debuggers has its limitations and lapses.

)) If the status register is explicitly modified at any part of the code which if faced while stepping in any mode in KDB, the debugger may loose control totally and behaves errorneously as KDB depends on trace bit of status register. (NB:- spl routines have been modified to take care of this , as they are encountered quite frequently.) To avoid this problem if one adds such a statement, one should take care of the fact that the trace bit (ti i.e msb of the status register) before modification should be or'ed with the new status register value.

)) In case of forking of processes through the debugger there is a

possibility of malfunctioning.

)) In case of routines like sleep or wakeup where there is explicit context switch or stack saving or changing, KDB cannot behave properly in those places.

)) Breaks at h/w clock service routines should be avoided as they are executed very frequently.

)) Breaks within the debugger code must be avoided.

)) Breaking / stepping etc. are done through the interrupt service section of the kernel code. A C funtion named "trap" takes care of proper operation at the initial phases of the interrupt service for most of the interrupts including the trap used by KDB. So in case of breaks at and below the level of the C funtion "trap" the debugger fails.

)) KDB uses the block device i/o strategy function ("sdstrategy" now) and all functions called by it for performing i/o from its private filesystem. So breaks at these routines can create an infinite loop of break and its service, hence causing stack overflow.

SECTION II

Installation

The debugger code should be resident in the same virtual space as the test code as discussed in section 1. So during compilation of the iitkix executable code, one must add the kernel level debugger's source files.These files (names given below) reside in $BASE area if they are .h files and $BASE/src directory in case of .c & .s files.

a) **KDB File Set** :- You must have the following files in your directory of iitkix source and help files-

new .c files

1. *kdb.c*; 2. *disass.c*; 3. *hash.c*; 4. *symtab.c*; 5. *parse.c*;
6. *bkpt.c*; 7. *showline.c*; 8. *sysc.c*; 9. *alias.c*.

old modified .c files

1.*main.c*; 2. *prf.c*; 3. *ntrap.c*; 4. *bio.c*; 6. *clock1.c*.

new .s files

1. *ch_st6.s*.

old modified .s files

1. *start.s*; 2. *spl.s*; 3. *vmstart.s*; 4. *save.s*; 5. *call.s*.

new .h files

1. *kdbcns.h*; 2. *kdbresource.h*; 3. *kdbmacro.h*; 4. *kdbfilsys.h*;
5. *kdbmain.h*; 6. *kdbsymtab.h*; 7. *kdbshowline.h*; 8. *bkpt.h*;
9. *disass.h*; 10. *symtab.h*;

Get these files from $BASE area and in case the user has also done some modification of the mentioned old files, he should make sure not to omit the modifications done in $BASE area (the portions modified are clearly marked).

b) **Compile the files** :- Consult iitkix.mk in $BASE/src area for this part if you wish to use the makefile facility. Compilation must be done with **'-g' , '-N' and '-DKDB'** options. The '-g' option will enable the source level debugging facility of the debugger and '-N' option is required for making the kernel version to be modifiable when loaded in memory. The '-DKDB' option is required to compile the changes made in the old C files which are put under conditional compilation flag *'KDB'*, for successful working of KDB. Finally link the .o files with "-e _start -T 0x10008000" to set the entry point of the o/s code at offset 0x10008000 and to assign this

virtual address to the start routine. NB:- Some of the source files you may not be interested in for debugging and you may not compile them with '-g' option, debugger may get confused if you want to interact in terms of source lines at those regions as it does not have the necessary symbolic information for this region.

c) **Create the simple file system containing your C source files :-**

A partion of disk /dev/sdxx (now /dev/sd2b) should be used for the filesystem. In $BASE/src directory there is a utility called "creatfs"(source code available in creatfs.c). Copy that to your src directory. For the first time to creat the filesystem you have to do-

creatfs <list of all your C source files used for making iitkix executable file, preferably in the same order with which their corresponding obj (.o) files are linked>.

Consult fs.mk in $BASE/src area to find out what is to be done as an example. This will essentially creat a simple file system on /dev/sdxx of the format described in section 1. Once the filesystem is formed, on the process of debugging one may like to change the code in some of the files or add a set of new files in the set. Any updatation should be reflected in the /dev/sdxx filesystem by

creatfs -m <modified or new file names>

which tells the utility to grow or shrink or add a file in the file system.If the growth in any file is so large that it cannot be accommodated in the 2 block gap then the utility indicates an **OVERFLOW**. In this case you are expected to rebuild the file system as mentioned earlier. If you find that the number of files are so large that the header cannont accommodate entries for all the files, change the constant #HEADERSIZE in the help file kdbfilsys.h. Do not forget to recompile the debugger files as well as creatfs.c (do "cc -o creatfs creatfs.c" to rebuild the utility). You must ensure that the /dev/sdxx filesystem is up-to-date when you boot iitkix for debugging as the debugger reads from the /dev/sdxx partition to show lines etc.

d) **Booting with KDB:-**

After you boot with your version of iitkix the debugging option can be set when the following question appears on the console at boot time -

"Do you want to invoke kernel debugger(y/n)-"

An word of caution at this point of time, don't say 'y' until you are sure that the latest version of your iitkix executable file is in the root of IITKIX FILE SYSTEM because the debugger will read from this executable file and set up its internal tables. If this file is not the same as what you are booting with debugging option set, all the address and symbolic refences will not correspond with the current working version of the kernel and the debugger will function incorrectly. NB:- If a file resides in the root of BSD4.2 file system it doesn't mean it is in the root of IITKIX file system. To transfer any file from BSD4.2 to IITKIX filesystem the following should be done :-

In BSD4.2 mount any scratch partition (like /dev/sd2f) with a BSD4.2 file system already existing there on some mount point (like /u1,say). Copy your version of iitkix to that and do ls -i to note the inode no of the copied iitkix. Then you shutdown and boot iitkix(it is the executable version lying at the root of BSD4.2 file system). When the above mentioned question will appear on the console you say 'n'(If you are not sure about your version guaranteeing an error free booting, use an error free version of iitkix for transferring of iitkix). After the login shell appears on any of the ttys login as root and say-

42to3 (devname) (inode no) (filename in IITKIX file system)
(/dev/sd2f) (the no you) (iitkix)
(are supposed)
(to note down)

This '42to3' (in /bin of IITKIX file system) is a facility for tranferring any file from BSD4.2 format to SYSTEM III (i.e IITKIX) format. Don't forget to *'sync'* a good no of times before you hit reset again. The *'sync'* command ensures that all the disk buffer's containing modified informations made by the user with the correspnding disk version not updated (DELAYED WRITE), will be flushed back in the disk and hence keeps the filesystem consistant. Now you boot your error prone version and say 'y' when the option comes. During booting KDB will read from the iitkix residing at the root of IITKIX filesystem and initialise all its internal tables indicating on the console which all files it is processing. After the initialisation phase is over the debugger goes into an interactive mode where you are advised to set a set of breakpoints

in the region of the code you want to debug.The option list will always be displayed on the console whenever the debugger is waiting for a command from the user in interactive mode. During the first phase all the options (as mentioned later in the manual) are not valid and on the option list on the console only the valid options appear (for e.g single or line stepping are not valid option during this phase).

SECTION III

Debugger manual

The debugger can be operated at assembly level (like *adb*) and somewhat at the source level (like *DBX*). The command set on the otherhand is similar to *sdb*.

i) When you are following the assembly level convention then an address in the code segment can be specified as :

1. A hex address preceded by prefix 0x,0X or # (e.g 0x0FFEE200, #10008000).
2. A decimal or octal absolute value with prefixes 0t/0T and 0o/0O respectively (e.g 0T10,0o12).
3. A function name in accordance to the assembly convention (for e.g _main) which is _(function name in your C code) in most of the cases.
4. A function name with offset. The syntax is -

(funcname)'+'['#','0'['x','X'],'0'['o','O'],'0'['t','T'],]offset

The last empty option signifies if you don't provide any prefix it will be accepted as a hex number (e.g. _main+#A).

NB:- The value more than 9 hex and less than 10 hex should be represented as A,B,C,D,E and F (i.e in capital letters). In this manual frequently the words ***valid addr_form*** and ***assembly addr_form*** will be used. Any of the above mention address forms is a valid addr_form and among them address expressions of the forms 3 & 4 are assembly addr_form.

ii) Reference to C source lines are made in the form

'L'(lineno)['/'(filename)]

where if the option filename is not specified, the 'current'

★★ The conventions followed in this manual for showing the syntax of any argument or command are-

- Any printable character will be referenced as '(char)'
- Any optional part will be refered as [(optional part)] if there is a single option which can be ommited. If there are multiple options then the set of allowable options will be indicated within a pair of square bracket delimited by *commas* like [(option#1),(option#2),.....,(option#n)]. If the last option is empty,then the option part can altogether be ommited and otherwise not.

file is assumed (see later for definition of current file). This form will be termed as *line_form* in the rest of this section. Examples of valid line_forms are L31 and L20/pipe.c.

iii) Variables in the data segment can be accessed in the assembly as well as in source level conventions. For example in a structure definition like

```
struct xxx{
    int     a_field;
    short   b_field;
     ...

} var;
```

_var+#4 will signify var.b_field at assembly level whereas in source level it can be referred to as `var.b_field` directly.

Any variable reference is considered as global reference if it is immediately preceeded by (i.e without a gap in between) the character 'G'. Otherwise it is taken as local reference. The only exception is the command 'W' where if the argument is preceeded by 'S' or 'F' they refer to structure and function name respectively. Subsequently any variable reference will be termed as *global_var_form* if it preceeded by 'G' else it is a *local_var_form*. An large set of commands are provided in KDB. Almost all the commands take one or more arguments which if not specified will be prompted for by KDB (see the test run for example). A typical command line looks like -

(command character)[' ','\t']+ [(argument set with no embedded white space)]

The commands are described below.

COMMANDS

1. V :: Produce the disassembled code starting form a valid code addr.

The argument is either a valid address or a source lineno signifying the start of the code to which it has got translated. The number of instructions to be disassembled can be provided directly at the end of the argument string delimited by a comma.If not provided the debugger will ask for it subsequently. The valid syntax for the argument is one of the following

- ***empty*** (in this case debugger will ask for the start address and no of instructions to be disassembled)
- ***<valid addr>['','<no of instr. to be disassembled>]***
- ***<line_form>['','<noof instr. to be disassmbled>]***
- '— ' (this option disables the automatic display of disassembled code at the location of break/step)
- '+ ' (this option enables the automatic display of disassbled code on break/step)

NB:- For the rest of the manual the following conventions will be followed addr_form (=) (valid addr) (the syntax of valid addr is described at the beginning of the debugger manual)

line_form (=) 'L'(lineno)['/'(filename string)]

One word of caution: if the start address for this command (and also command 2) does not really point to a valid instruction opcode one or more of the subsequent instructions may be faultily disassembled or KDB may write "CANNOT DISASSEMBLED".

Examples : v; v _main+2; v L10/main.c,20 etc.

2. V :: Produce the disassembled code along with source lines (if any). Argument is of the same format as that of option 1, only difference being that the options '—' and '+' respectively disables and enables both automatic disassembly and C source line display (if the break/step loation corresponds to a line) on break /step.

Example : V _main,10 etc.

3. f :: Get the hex address corresponding to an assembly addr_form.

Examples : f , f _pipe+10 etc.

4. a :: Get the assembly addr_form string given a hex address.

Inverse of the 'f' command.

Examples : a , a 10008000 etc.

5. F :: Get or Change currently set filename.

If no arguments are provided with this command the debugger will only inform you about the currently set filename and will not ask for any input unlike

other commands. To change the currently set filename you must provide it in the argument itself. To start with the debugger gets initialised to a default file which is the C source file the object module of which is linked first (check up your iitkix.mk) . With this command you can override that default value. All the line number references (and line_forms without filename) will refer to the lines of currently set file name.

Examples : F (will show currently set file name) ; F sd.c (will set current file to sd.c (if present in file system)).

6. l :: Display lines from C source code

KDB reads from private file system[ref section 1]. The arguments can be

- ***empty*** - the debugger will ask for a proper argument
- ***a integer signifying lineno*** - the line if valid from the currently set file will be shown.
- ***<lineno1>'.''.'<lineno2>*** - the range will be shown from the currently set file. Remeber "lineno"s signify unsigned integers only and not line_form.
- ***function name as in source code*** - the first 10 lines of the function will be shown (filename will be indicated).
- '– ' - Disables the automatic line display on break/step.
- '+' - Enables the auto-line-display facility.

Examples : l , l 10 , l 10.30 , l bread.

7. b :: Set a breakpoint .

The argument can be

- ***empty*** - debugger will ask for input.
- ***addr_form***
- ***line_form*** - if filename is not provided currently set filename is taken.

On a break at a point which lies in the code range of a C source funtion, the current filename is set to the file where the funtion resides.

Examples : b, b 0x10008000, b _pipe+#3, b L10 etc.

NB :- A wrong starting address can lead to havoc. So please check up through command 'v' if you are providing addr_form and you have a doubt.

8. B :: Show all the currently set break points.

Takes no arguments. The form in which the break points are shown is as

follows -

⟨serial no. of the break point⟩ ⟨function_name⟩'+''#'⟨offset⟩ 'f''f' ⟨filename where the function resides in C source_code⟩'/''L'⟨lineno★⟩

The star(★) in the lineno field signifies the fact that you may set a break point at a location that doesn't correspond to any source line as it will be if it has an assembly code as origin or just may lie in between a set code a source line gets translated to. In that case ⟨filename⟩/LO::UNKNOWN will be displayed.

9. d :: Delete a Break point.

The argument is just the serial number as one gets through command number 8. If no argument is given the debugger will ask for the serial number.

Examples : d, d 2 etc.

10. D :: Delete all the breakpoints currently set.

Takes no argument.

11. s :: Single step at assembly level.

Takes no argument. The current assmbly instruction to be executed immediately will be shown.The assembly addr_form of the current address will also be specified. If the current addr corresponds to the start of the set of code which a C source line gets translated to ,then that line will also be shown. If on the process of single stepping the execution jumps to a location which corresponds to code in a different file the currently set file will be automatically set to that file.

12. S :: Step multiple number of times at assembly level.

Argument should be a unsigned integer specifying the number of times you want to step without halting. If no argument is provided the debugger will ask for it.

Examples : S, S 4 etc.

NB:- If no of steps specified is found to be too large then KDB mentions that and steps only once.

13. L :: Step C source line-wise.

The arguments can be

- ***empty*** - like 'n' in DBX.
- ***['-'['u','U']][(lineno)]*** - step upto line number specified. If line number is not specified KDB asks for it. This option is like "go -until lineno" in Appolo's DEBUG debugger.

If argument is empty KDB steps in terms of line in the current procedure and when a return from the procedure is reached it automatically stops at the next line of the calling function (if any, beware here if return from the current C routine takes you back in a assembly routine). The currently set file is also changed as the line stepping traverses through functions of different files. For example if you start line stepping from a break in the routine "pipe" in file pipe.c which is called from a function "trapi" in ntrap.c file, then as long as you are in the pipe routine the current file will be pipe.c (if not explicitly changed by the user) and on a return from that routine the current file will change to ntrap.c provided the return is faced during line stepping . The same is also true for other two forms of stepping discussed earlier. Another valid argument can be -[uU]<lineno>, no other options are yet supported. This option signifies go upto line specified in argument without halting. The line should lie within the same current C function otherwise the debugger will deny to recognise this command, displaying proper error message. A word of caution at this point of time. If there is a control flow statement in between the line from which one is starting linestepping in this mode and the destination line specified in argument; and if there is a possibility that the destination statment will not be executed at all before the function is exited, one may not get back the control at all unless a break point is faced or another activation of the same function comes where it really reaches the desired lineno. So care should be taken in chosing the lineno. If one skips the lineno in the argument and only provides the option -[uU], the debugger will ask for the lineno.

14. C :: Continue execution after break or single stepping or linestepping.

15. Q :: Quit the initialisation phase of KDB.

Takes no argumant. When for the first time the debugger will come up for interacting with you, all you are expected to do is to set a group of break

point and continue with the normal boot process untill a break is encountered. At this stage q is significant. The option list will be smaller at this initial stage as will be clear from the option list on the screen(it will grow when the debugger will get control after a break !).

16. r :: Show Register contents.

Takes no argument. The registers of MC68020 is shown along with status register and user stack pointer. (NB:- As the debugger resides in kernel code ,it uses the supervisory stack which can easily be verified by checking whether the Supervisor bit of the status register is on or not).

17. W :: Display the call sequence.

The activation tree will be displayed. The following convention is followed in showing the tree. If there are any C function found alive in the tree, the line last executed in that function is shown along with the filename wherethe function is declared. Also the whole set of parameter values along with their name is displayed as shown below-

$Line ⟨lineno⟩/⟨file name⟩:⟨function name⟩(⟨param#1name⟩=⟨value⟩,...,⟨param#n⟩);;

On the otherhand the non-C routine's presence in the activation tree is shown as-

$—/—:⟨routine name in assembly_form⟩

One common mistake should be kept in mind : almost all the C functions gets translated to m/c code which if disassembled contains the following sequence of instructions to begin with

```
        bra xxx
yyy:    .....
        .....
        unlk fp
        rts
xxx:    link fp,⟨val⟩
        bra yyy
```

the link instruction should be executed (by stepping for e.g) for getting a proper call sequence. (Similar to "where" in DBX)

18. t :: Stack trace in m/c level.

The (supervisor) stack(long word wise) and the current frame pointer are displyed.

19. p :: Print value of an expression.

The arguments can be of following type (remember the rule of global/local reference)

- ***empty*** - The debugger will ask for inputting proper expression which can any one of the following forms.
- ***['G']Variable name of basic or user defined structure type*** - if the variable stands for any user defined structure, value of all fields of the variable will be printed. If there is any union in the structure, KDB cannot find out for which one the value actually stands. So it prints the value in all the fields of the union and it depends on the user to properly interprete it. Individual fields can be accessed in the same syntax as in C.
- ***['G']Variable name of pointer type*** - in case of any pointer if it is char pointer the string it points to will be printed. otherwise the address it points to will be printed (in hex) with 0x preceeding it.
- ***['G']#Pointer name[<subexpression for valid fields and subfields in C syntax>]*** - if the pointer points to a basic datatype the value of that location will be printed or else if it points to a composite datatype like structure/union then all the fields will be printed as in 2nd case.
- ***['G']&Variable name[<subexpression for valid fields and subfields in C syntax>]*** - the address of any location can be found out by &<expression>. If it is a local or parameter and is placed in a register the register name will be printed.
- ***['G']Array name[<index expression>][<subexpression for valid fields and subfields in C syntax>]]*** - in case of an array if the array name is typed the whole array will be printed (take care in case of big array!) and individual element can be accessed through <arrayname>[<indx>] format as usual. The unique feature provided here is that you can see few consequitive elements of an array by typing <array name>[<loval>..<hival>] provided you don't violate array limits. Also <array name>[ALL]<expression signaifying fields or subfields> will print the relevent information in the whole array. For example if you have a array declaration

```
struct xxx {
        .....
        struct yyy{
                ....
                char *name;
                ....
        } y_info[YSIZE];
        ....
}x_table[XSIZE];
```

then to see the name of all the y_infos in the range of index 'A' and 'B' from x_table type "p **x_table[A..B].y_info[ALL].name**" whereas to see the name of all Cth instances of name in the set of y_infos throughout x_table type "p **x_table[ALL].p_info[C].name**".

- ***O['x','X'] hex address*** - you can give an hex addr as 0x(hex val) as expression and mention whether byte/word/long word is to displayed by following it with /b , /s or /l respectively (default is short). NB:- the expression to be evaluated can be of the form just as in DBX with the above mentioned set of extentions.This option is similar to "p exression" in DBX.

20. P :: Traverse a linked list data structures.

With this instruction one can follow a linked list and print the fields of the structure of the linked list units. The arguments can be

- ***empty*** - the argument will be asked as usual
- ***<expression#1>'^'<expression#2>':'[<expression#3>]',<expression#4>*** - where the expressions can have syntax and meaning as described below.

★ <u>expression 1</u> : it must be a name that signifies a member of the linked list. For example if there is a linked list of structure "xxx" and "p_xxxtype" is a pointer to a location "xxxtype" which is of structure type "xxx", and if "xxxtype" is a member of the linked list then both "xxxtype" and "*p_xxxtype" is a valid expression for this field. NB:- A global reference must be explicitely mentioned by a 'G' as the first letter of this expression, otherwise it will be interpreted as a local reference.

★ <u>expression 2</u> : it must be a field name of the structure type of individual member of the linked list. Also it must point to the structure of same kind. For example in a declaration

```
struct xxx {
        struct xxx *x_forw;
        struct xxx *x_backw;
        int a_field;
        short b_field;
        ...
        struct yyy *y_ptr;
};
```

both "x_forw" and "x_backw" are valid names for this field but "y_ptr" is not.

★ expression 3 : this has got a syntax

if ⟨f_name⟩ ⟨=⟩ ⟨any valid field name of the struct of linked list members⟩

then

```
⟨expr 3⟩: empty
        | ⟨expr 3a⟩
        ;
```

where

```
⟨expr 3a⟩: ⟨f_name⟩
        | ⟨f_name⟩.⟨expr 3a⟩
        ;
```

For example with the above mentioned declaration of struct "xxx" "a_field", "a_field.y_ptr.b_field" etc are valid entry for this field. If this field is kept empty all the fields will be displayed.

★ expression 4 : this field should be an unsigned integer signifying the number of links to follow for printing. If on following the link a NULL pointer is found KDB delivers the message and aborts the process of following further. NB :- Any error in the set of fields is clearly indicated.

Example :- with the previous description of struct "xxx" and variable "p_xxxtype" P *p_xxxtype^x_forw:a_field.x_forw.y_ptr.20 and P xxxtype^x_backw:10 are valid commands.

21. W :: Display description of a type-specifier or inform certain variable/function's type.

This command shows the qualification of any structure or variable or a C function. This is a facility similar to whatis in DBX. The argument can be of the following form-

a)S<structure name> :- All the subfields name and types will be indicated as they appear in a C structure enumeration. Only defined typedefs will come in extended format.

b) <global_var_form> (i.e G<global variable or C syntaxed expression signifying subfields of a variable if applicable>)

c) <local_var_form>

d) F<function name> :- The function type will be printed with function parameters. The types of the parameters will also be specified with their names.

For e.g. if it is declared

```
struct xxx {
            ...
        union {
            int yyy;
            struct {
                ....
                char zzz[35];
                ....
            } www;
          ....
          ....
      }x_un;
    ...
} name[10];
```

then "**W Gname[3].x_un**" will print the whole union defination where as "**W Gname[2].x_un.www.zzz**" will print *char name[2].x_un.www.zzz[0..34];* and if there is a function "funct1" which returns a pointer to structure "xxx" then "**W Ffunct1**" will print

```
struct xxx * funct1(p1,p2,....,pn)
    int p1;
    char p2;
    ......
    int (* pn)();
```

when the parameters are declared as shown in the source code also. NB:- By function the debugger understands C source functions only not any assembly routine. The anomalies described in section 1 are relevent here.

22. A :: Alias facility for extending command set of debugger dynamically.

The allowable arguments are

- *⟨alias command char⟩'='⟨original command char⟩'|'⟨argument of orginal command⟩* :- Alias command character can be any printable character which is not already in the command set of KDB or not already aliased. Only ten aliases can be set in the present implementation. For e.g. "A F=p|Gu.u_arg" is an invalid alias as F command is used for showing and setting the currently set files as explained previously. But "A z=p|Gu.u_procp->p_pid" is a valid command and once set z stands for the command for displaying the currently running processes.
- *'-'⟨option name⟩[⟨option argument⟩]* :- Through this one can see and delete the aliases set. Options supported are
  - ★ p,P : Display the current aliases. Takes no option argument.
    For e.g "A -p" or "A -P".
  - ★ d,D : Delete alias for the command character provided in the option argument.
    For e.g if "A -p" prints z = "p Gu.u_procp->p_ppid"
    Z = "P Gf_tab[0..1].p_info[ALL].pname"
    x = "S 10"
    then after a "A -dZ" command another "A -p" will print
    z = "p Gu.u_procp->p_ppid"
    x = "S 10"

SECTION IV

SAMPLE DEBUGGING SESSION

As mentioned earlier while booting a message comes on the console. If you say 'y' then the initialisation of KDB starts. Untill you quit the interactive mode the option list is smaller which grows only after a valid break.

```
KDB>>(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
W Sinode
struct inode {
    char i_flag;
    short i_count;
    short i_dev;
    unsigned short i_number;
    unsigned short i_mode;
    short i_nlink;
    unsigned short i_uid;
    unsigned short i_gid;
    int i_size;
    struct  {
          union  {
              int i_a[0..12];
              short i_f[0..25];
          } i_p;
          int i_l;
    } i_blks;
    struct locklist * i_locklist;
    struct sem * i_semptr;
    struct chan * i_chan;
}

kdb>>(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
W Ginode
struct inode inode[0..44];

kdb>>(v/l/p/P/a/W/F/f/m/b/d/D/B/s/A/q):-
W
Type in the Structure/variable name :-
G inode[0].i_blks.i_p
union  {
     int i_a[0..12];
     short i_f[0..25];
} inode[0].i_blks.i_p;


kdb>>(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
l hashi
File = hash.c
-> 8    {/* GIVEN THE GLOBAL SYMBOL NAME FIND THE
-> 9         INDEX INTO THE HASH TABLE */
-> 10
```

```
-> 11   char c;
-> 12   int i;
-> 13   int sum = 0;
-> 14           for(i =  0;s[i] != ' ';i++)
-> 15                   sum = sum + (s[i] - '0');
-> 16                   dummy();
-> 17           return((sum % HASH1BASE));
-> 18   }
kdb))(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
v
GIVE THE STARTING ADDR(in procname + offset form)
_hash1
Give The No. of Instructions To Be Disassembled
3
_hash1+#0               bra   _hash1+#50
_hash1+#2               clr.l    (0xfffffff6,a6)
_hash1+#6               clr.l    (0xfffffffa,a6)

kdb))(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
V _hash1+2,5
-> 13   int sum = 0;
_hash1+#2               clr.l    (0xfffffff6,a6)
-> 14           for(i =  0;s[i] != ' ';i++)
_hash1+#6               clr.l    (0xfffffffa,a6)
_hash1+#a               bra   _hash1+#2a
-> 15                  sum = sum + (s[i] - '0');
_hash1+#c               mov.l   (0x8,a6) ,   a0
_hash1+#10              add.l    (0xfffffffa,a6), a0

kdb))(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
F pipe.c

kdb))(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
F
Currently set file is pipe.c

kdb))(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
b L16
kdb))(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
b
Give the break point specification :-
_falloc+#5C
(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
B
THE BREAKPOINTS ARE
-------------------------
[1]-> _falloc+#5C - #1000B69C || fio.c/L0::UNKNOWN-(
[2]-> _pipe - #1000E72A || pipe.c/L16 -(

kdb))(v/V/l/p/P/a/f/W/F/b/d/D/B/A/q):-
q
OK! HOPE YOU HAVE SET A BREAK POINT
```

At this point of time the kernel code goes back for normal boot process. Let us assume that a system call is made by an user process which in turn calls pipe routine of kernel code. Then the debugger gets back control through a break. The next phase of sample debugging session is presented below.

```
->>>Break at _pipe[0x1000E72A] ***
->16 {
_pipe+#0        bra _pipe+#B6

kdb>>(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
v -
kdb>>(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
L
->>Stopped at _pipe+#2[0x1000E72A] ***
->21 ip = ialloc(rootdev);

kdb>>(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
A  z=p!Gu.u_procp->p_pid

kdb>>(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
L -u25
->>Stopped at _pipe+#20[0x1000E74A] ***
->25 wf = falloc();

kdb>>(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
A D=p!Gu.u_arg[2]
ERROR:- Debugger command 'D'

kdb>>(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
V +
kdb>>(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
s
->>Stopped at _falloc[0x1000B640] ***
->103 {
_falloc+#0    bra  _falloc+#5C
kdb>>(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
S
no. of steps = 3
->>>Break at _falloc+#5C[0x1000B69C] ***
_falloc+#5C     link  a6,0xfffffff8

kdb>>(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
A  -p
"z   =   p    Gu.u_procp->p_pid"

kdb>>(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
S 3
->>Stopped at _falloc+#2[0x1000B642] ***
->291  if((i = ufalloc(0)) < 0)
_falloc+#2      pea #0x0
```

```
kdb))(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
z
1024

kdb))(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
p file[0..5]
[0x10504380 0x10510300 0x0 0x0 0x0 0x0]

kdb))(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
P Gbuf[0]^b_forw:b_flags.b_bcount,3
1)) b_flags = 9
    b_bcount = 1
2)) b_bflags = 3
    b_bcount = 2
3)) b_flags  = 11
    b_bcount = 0
kdb))(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
S 10
-)) Stopped at _ufalloc+#0[0x1000B62A] ***
-)267 for(i=0;i(NOFILE;i++){
_ufalloc+#2  mov.l (a6,0x8), d7

kdb))(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
w
THE CALL SEQUENCE :-
-----------------------
$Line 267/fio.c : ufalloc(n=0);;
$Line 291/fio.c : falloc();;
$Line 25/pipe.c : pipe();;
$Line 328/ntrap.c: trap1(f=0x1000E72A);;
$Line 182/ntrap.c: trap(dev=0,d0=120897,d1=0.....ps=10975,pc=2103743);;
$Line ---/--- : syscall+#9A

kdb))(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
p i
0

(v/V/l/p/P/a/W/F/f/b/d/D/B/r/w/t/s/S/L/A/c):-
p G*b_sel
ERROR : Illegal Reference Through NULL Pointer.
```

One can expect to find out bugs through this kind of session.

BIBLIOGRAPHY

[ACC86] Mike Accetta et. al, *Mach : A new Kernel Foundation for UNIX Development, Usenix Conference, pp93-112,* Summer 1986.

[ADA86] E. Adams and S. S. Muchnick, *Dbxtool : A Window Based Symbolic Debugger for SUN Workstations,* Software — Practice and Experience, Vol.16(7): pp653-669, July 1986.

[AHO77] A.V. Aho and J.D. Ullman, *Principles of Compiler Design,* Addison-Wesley, 1977.

[APMN87] Apollo Computer Inc., *DOMAIN Language Level Debugger Reference,* 1987.

[BAC86] M.J. Bach, *Design of UNIX Operating System,* Prentice Hall of India,1986.

[BAR87] G. Barua, *IITKIX: A UNIX+system III compatible OS Kernel,* CSI Journal, Vol. 17(2): pp21-28,1987.

[BHA87] S. Bharati, *Enhancements and Performance Monitoring of UNIX V6 on S-32,* M. Tech Thesis, IIT. Kanpur, 1987.

[CHA82] Chandrashekar et. al, *DISASSEMBLERS - A Tool for the Analysis of Assembled Code,* XVII Annual Convention, CSI, Vol. 2: pp14-20, Jan 1982.

[CRA86] William Crammer and Gerry Kane, *68000 Microprocessor Handbook,* McGraw-Hill Pub. (2nd ED), 1986.

[HAM83] Dick Hamlet, *Debugging "LEVEL" : Step-wise Debugging,* SIGPLAN Notices, Vol. 18(8) : pp4—8, Aug. 1983.

[GRA83] W.G. Gramlich, *Debugging Methodology,* SIGPLAN Notices, Vol. 18(8) : pp1— 3, Aug. 1983.

[HEN82] J.L. Hennessy, *Symbolic Debugging of Optimised Code,* ACM Trans. on Programming Languages and Systems, Vol. 5(3) : pp323–344, July 1982.

[JON83] J.D. Johnson and G.W. Kenney, *Implementation Issues for a Source Level Symbolic Debugger,* SIGPLAN Notices, Vol. 18(8) : pp149–151, Aug. 1983.

[KEN88] B.W. Kernighan and D.M. Ritchie, *The C Programming Language,* Prentice Hall of India, 1988.

[LEF88] Leffer et. al, *The Design and Implementation of The 4.3BSD UNIX Operating System,* Addison-Weslay Pub., 1988.

[LEV87] L.A. Leventhal et. al, *680000 Assembly Language Programming,* McGraw-Hill Pub. (2nd ED), 1987.

[MAR86] J.F. Maranzano and S.R. Bourne, *A Tutorial Introduction to ADB,* Language Processors Reference Manual (Ver 1.0), HCL India, 1986.

[MOT85] Motorola, *MC68020 32-Bit μP Handbook,* McGraw-Hill Pub. (2nd ED), 1986.

[MYR78] G.J. Myers, *The Art of Software Testing,* Wiley-Interscience Pub., 1978.

[SEK88] A. Sekar, *DBG : A Source Level Debugger for C,* M. Tech Thesis, IIT. Kanpur, 1988.

[UNI83] UniSoft Systems, *'C' Interface Notes For 68000 UNIX Systems,* Programming Tools, Uniplus Reference Manual, 1983.

[UMNI86] Unix Programmer's Manual (for Horizon III), Part I & IV, HCL India, 1986.

[UMNM88] Unix Programmer's Manual (for Magnum), HCL India, 1988.

[WIN88] Russel Winder, *JDB : An Adaptable Interface for Debugging,* Software – Practice and Experience, Vol. 18(3) : pp221– 238, Mar. 1988.